

## Chapter 12

### 12.1 PROCESSOR ORGANIZATION (প্রসেসরের গঠন)

#### ◆ প্রসেসর কী করে?

প্রসেসরের মূল কাজগুলো ৫টা ধাপে করা হয় —

1. **Fetch instruction** → মেমরি (register, cache, main memory) থেকে নির্দেশ (instruction) নিয়ে আসে।
2. **Interpret instruction** → সেই নির্দেশটা ডিকোড করে বুঝে নেয় কী কাজ করতে হবে।
3. **Fetch data** → প্রয়োজনীয় ডেটা মেমরি বা I/O ডিভাইস থেকে নিয়ে আসে।
4. **Process data** → গাণিতিক বা লজিক্যাল (logical) অপারেশন করে।
5. **Write data** → প্রক্রিয়াকৃত ডেটা (result) আবার মেমরি বা I/O ডিভাইসে লিখে দেয়।

→ এগুলো করতে গেলে প্রসেসরকে কিছু তথ্য অস্থায়ীভাবে মনে রাখতে হয় — যেমন,

- শেষ কোন ইনস্ট্রুকশন এক্সিকিউট করল
- পরের ইনস্ট্রুকশন কোথায় আছে
- বর্তমানে কোন ডেটা নিয়ে কাজ হচ্ছে

তাই প্রসেসরের ভিতরে ছোট্ট একটা মেমরি ইউনিট থাকে, যেটাকে বলে Register।

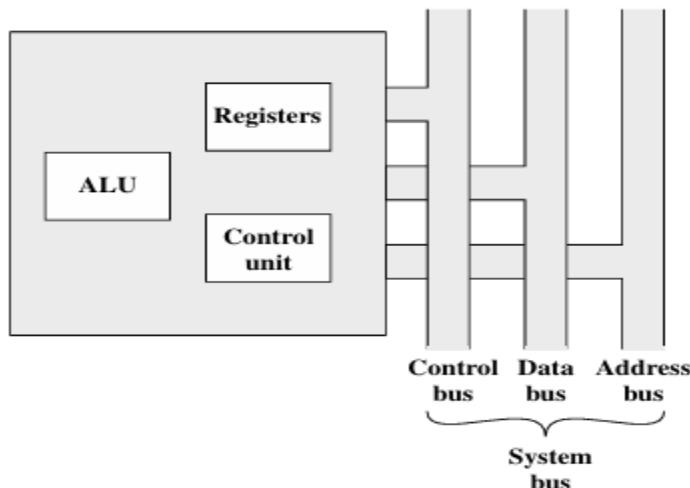


Figure 12.1 The CPU with the System Bus

## ⚙️ প্রসেসরের প্রধান উপাদানসমূহ

প্রসেসরের ভেতরে প্রধানত ৩টি অংশ থাকে:

উপাদান	কাজ
<b>ALU (Arithmetic and Logic Unit)</b>	গণিতিক ( $+$ , $-$ , $\times$ , $\div$ ) ও লজিক্যাল (AND, OR, NOT, COMPARE) কাজ করে।
<b>CU (Control Unit)</b>	ডেটা ও ইনস্ট্রাকশন কোথা থেকে আসবে, কোথায় যাবে—সবকিছুর নিয়ন্ত্রণ করে।
<b>Registers</b>	অস্থায়ীভাবে ডেটা, ইনস্ট্রাকশন, বা ঠিকানা (address) সংরক্ষণ করে।

▣ এই অংশগুলো পরম্পরের সাথে Internal Bus দিয়ে যুক্ত থাকে (data path)।

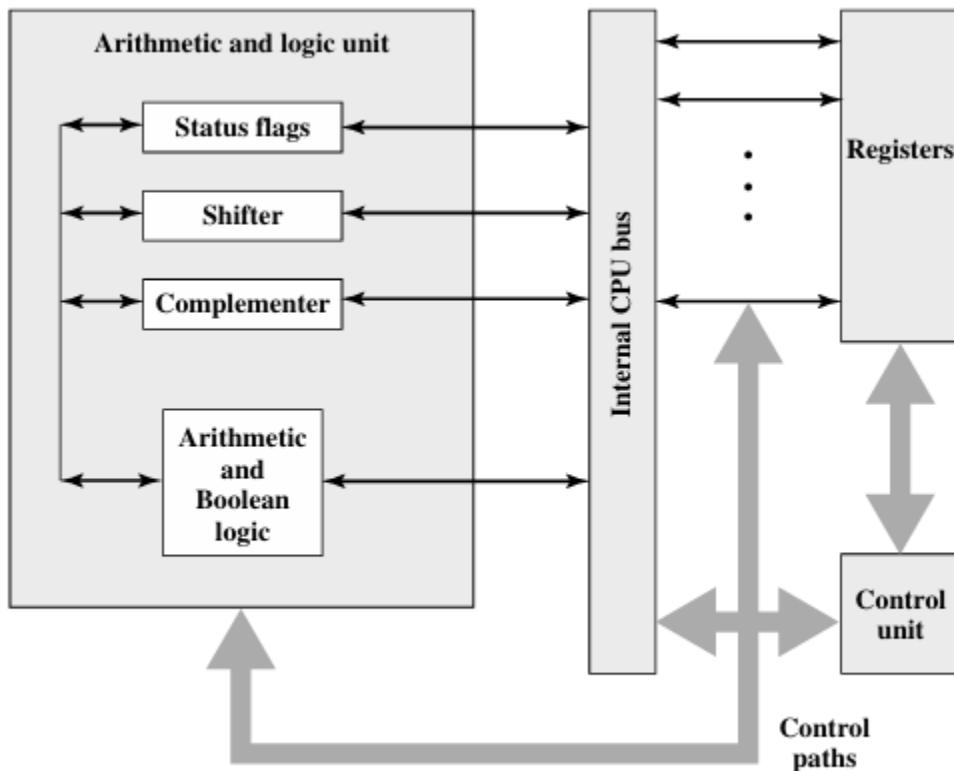


Figure 12.2 Internal Structure of the CPU



## Registers: প্রসেসরের ভিতরের ছোট মেমরি

Registers হলো প্রসেসরের সবচেয়ে দ্রুত মেমরি।

এগুলো মূলত দুই প্রকারের:

1. User-visible registers → প্রোগ্রামার সরাসরি ব্যবহার করতে পারে
2. Control and Status registers → সিস্টেম ও অপারেটিং সিস্টেম দ্বারা নিয়ন্ত্রিত



### 1 User-visible Registers

এই রেজিস্টারগুলো প্রোগ্রামার ব্যবহার করে —

এগুলো চার প্রকার হতে পারে:

প্রকার	কাজ
General Purpose	সাধারণ ডেটা রাখে, সব কাজের জন্য ব্যবহার করা যায়।
Data Registers	শুধুমাত্র ডেটা রাখার জন্য ব্যবহৃত হয়।
Address Registers	ডেটার ঠিকানা (address) রাখার জন্য ব্যবহৃত হয়।
Condition Code (Flag) Registers	বিভিন্ন অপারেশনের ফলাফল (positive, negative, zero, overflow ইত্যাদি) রাখে।



### Address Register-এর বিশেষ ধরণ

ধরণ	কাজ
Segment Register	Segmented memory system-এ segment-এর base address রাখে।
Index Register	Indexed addressing mode-এ ব্যবহৃত হয়।
Stack Pointer	Stack-এর উপরের দিক (top) নির্দেশ করে। Push/Pop এর সময় স্বয়ংক্রিয়ভাবে ব্যবহৃত হয়।



### Design Issues (ডিজাইন সংক্রান্ত সিদ্ধান্ত)

#### 1. General-purpose vs Specialized registers

- ০ সব রেজিস্টারকে general-purpose করলে flexibility বেশি, কিন্তু instruction-এর bit usage বাড়ে।
- ০ Specialized করলে bit usage কমে, কিন্তু flexibility কমে।

#### 2. Number of registers

- গবেষণা অনুযায়ী, 8 থেকে 32 রেজিস্টার হলে performance সবচেয়ে ভালো।
- কম হলে বেশি memory access লাগে, বেশি হলে তেমন লাভ হয় না।

### 3. Register Length

- Address register → ঠিকানার জন্য যথেষ্ট বড় হতে হবে।
  - Data register → বিভিন্ন data type ধারণ করতে সক্ষম হতে হবে।
- 



## Control & Status Registers

এই রেজিস্টারগুলো user সাধারণত দেখতে পায় না, তবে প্রসেসরের কন্ট্রোল ইউনিট এগুলো ব্যবহার করে।

এগুলো প্রসেসরের অবস্থা (status), program counter, instruction register ইত্যাদি ধারণ করে।

---

## Condition Codes / Flags

Condition code বা Flag register-এ বিভিন্ন স্টেটাস বিট (status bits) থাকে যা অপারেশন শেষে সেট হয়, যেমন:

ফ্ল্যাগ	অর্থ
Z (Zero)	রেজাল্ট যদি 0 হয়
N (Negative)	রেজাল্ট যদি negative হয়
C (Carry/Borrow)	গাণিতিক অপারেশনে carry/borrow হলে
V (Overflow)	রেজাল্ট register capacity ছাড়ালে

→ এই ফ্ল্যাগগুলো conditional branching এর সময় চেক করা হয়  
যেমন: JZ (jump if zero), JC (jump if carry) ইত্যাদি।

---



## Condition Codes-এর সুবিধা ও অসুবিধা(EXAM)

সুবিধা	অসুবিধা
1 Arithmetic ও data movement instruction-এর মাধ্যমে স্বয়ংক্রিয়ভাবে সেট হয়, ফলে আলাদা COMPARE বা TEST দরকার কমে।	1 হার্ডওয়্যার ও সফটওয়্যার জটিল হয়।

সুবিধা	অসুবিধা
2 Conditional instruction যেমন BRANCH সহজ হয়।	2 Condition code আলাদা hardware path লাগে।
3 Multiway branch সহজ হয়।	3 কিছু বিশেষ কাজের (bit check, loop control) জন্য আলাদা instruction লাগেই।
	4 Pipeline-এ synchronization সমস্যা হয়।

## Subroutine call-এর সময় register handling

- কিছু প্রসেসরে subroutine call করলে সব user-visible register **স্বয়ংক্রিয়ভাবে save** এবং **restore** হয়।
- আবার কিছু প্রসেসরে প্রোগ্রামারকেই হাতে করে (explicitly) register save/restore করতে হয়।

## সারসংক্ষেপ:

বিষয়	ব্যাখ্যা
Processor-এর কাজ	Fetch, Decode, Fetch Data, Process, Write
প্রধান অংশ	ALU, CU, Registers
Registers-এর প্রকার	User-visible, Control/Status
User-visible register-এর ধরণ	General, Data, Address, Condition
Condition code ফ্ল্যাগ	Zero, Negative, Carry, Overflow
ডিজাইন সিদ্ধান্ত	ক্রটগুলো register থাকবে, ক্রট লম্বা হবে, general না specialized
উদাহরণ	Stack pointer, Segment register, Index register

## Control and Status Registers কী?

Processor-এর ভেতরে কিছু বিশেষ রেজিস্টার থাকে যেগুলো processor-এর কাজ নিয়ন্ত্রণ ও পরিচালনা করে।

এগুলোকে বলা হয় **Control and Status Registers**।

👉 এগুলো সাধারণত **user-visible** নয়, অর্থাৎ প্রোগ্রামার সরাসরি এগুলো ব্যবহার করতে পারে না।

তবে **Operating System (OS)** বা **privileged mode**-এ থাকা প্রোগ্রাম এগুলো অ্যা�্যেস করতে পারে।

---

#### ◆ Instruction Execution-এর জন্য প্রয়োজনীয় চারটি প্রধান রেজিস্টার:

রেজিস্টারের নাম	কাজ
Program Counter (PC)	পরবর্তী যে instruction টা fetch করতে হবে, তার address রাখে।
Instruction Register (IR)	সদ্য fetch করা instruction রাখে। এখানে opcode ও operand decode হয়।
Memory Address Register (MAR)	কোন memory address থেকে data পড়তে বা তাতে লিখতে হবে, সেই address রাখে।
Memory Buffer Register (MBR)	মেমরি থেকে যে data পড়া হয়েছে বা ষেটা লিখতে হবে, তা সাময়িকভাবে ধরে রাখে।

◆ সব processor-এ MAR ও MBR নামে রেজিস্টার নাও থাকতে পারে, কিন্তু একই কাজ করার জন্য buffer বা temporary register থাকে।

---

#### 🧠 Instruction execution-এর সময় এই চারটি রেজিস্টারের কাজের ধাপ:

1. **PC → IR:**  
Program Counter পরবর্তী instruction-এর address দেয়।  
সেই address থেকে instruction মেমরি থেকে আনা হয় এবং IR-এ রাখা হয়।
  2. **IR Decode:**  
IR-এ থাকা instruction ডিকোড হয় (opcode, operand নির্ধারিত হয়)।
  3. **MAR ↔ MBR:**  
Data বা address MAR এবং MBR-এর মাধ্যমে মেমরি বা I/O-র সাথে আদান-প্রদান হয়।
  4. **ALU Process:**  
ALU ডেটা প্রক্রিয়াজাত করে (গাণিতিক বা লজিক্যাল কাজ করে)।
- 

#### ⚙️ Program Status Word (PSW):

এটা এক ধরনের **special control register**, যেখানে processor-এর বর্তমান অবস্থার তথ্য থাকে। PSW-এর মধ্যে সাধারণত নিচের status flags থাকে 

Flag	অর্থ / কাজ
<b>Sign (S)</b>	শেষ arithmetic operation-এর ফলাফল positive না negative তা জানায়।
<b>Zero (Z)</b>	যদি operation-এর ফলাফল 0 হয়, তাহলে সেট হয়।
<b>Carry (C)</b>	Addition-এ carry হলে বা subtraction-এ borrow হলে সেট হয়।
<b>Equal (E)</b>	দুটি মান সমান হলে সেট হয়।
<b>Overflow (V)</b>	যদি arithmetic overflow হয়, সেট হয়।
<b>Interrupt Enable/Disable (I)</b>	Interrupt চালু বা বন্ধ আছে কিনা জানায়।
<b>Supervisor (Mode Bit)</b>	Processor বর্তমানে User Mode নাকি Supervisor Mode-এ আছে তা জানায়।

#### Supervisor Mode:

এই মোডে কিছু privileged instruction চালানো যায়, যেগুলো সাধারণ ইউজার প্রোগ্রাম চালাতে পারে না (যেমন: মেমরি প্রটেকশন, I/O কন্ট্রোল ইত্যাদি)।

#### অন্যান্য Control ও Status Registers-এর উদাহরণ:

রেজিস্টারের নাম	কাজ
<b>Interrupt Vector Register</b>	কোন interrupt ঘটলে, তার handler-এর address রাখে।
<b>System Stack Pointer</b>	System stack-এর উপরের দিকের address রাখে (subroutine, interrupt ইত্যাদির জন্য)।
<b>Page Table Pointer</b>	Virtual memory ব্যবস্থায় কোন page table ব্যবহার হচ্ছে তা নির্দেশ করে।
<b>I/O Control Registers</b>	Input-Output device নিয়ন্ত্রণে ব্যবহৃত হয়।
<b>Process Control Block Pointer</b>	Running process-এর তথ্য যেখানে আছে, তার address নির্দেশ করে।

#### Design সংক্রান্ত কিছু গুরুত্বপূর্ণ বিষয়:

### 1. Operating System Support:

Processor designer OS কীভাবে কাজ করে তা বুঝে রেজিস্টার ডিজাইন করলে control সহজ হয়।

### 2. Registers vs Memory Allocation:

- Register-এ control data রাখলে কাজ দ্রুত হয় কিন্তু খরচ বেশি।
  - Memory-তে রাখলে খরচ কম কিন্তু গতি ধীর হয়।  
⇒ তাই designer-রা একটা trade-off করে — কিছু তথ্য register-এ, বাকিটা memory-তে রাখে।
- 

### সংক্ষেপে সারাংশ:

#### বিষয়

#### মূল কথা

**Control & Status Registers** Processor নিয়ন্ত্রণ ও অবস্থার তথ্য রাখে।

**৪টি গুরুত্বপূর্ণ Register** PC, IR, MAR, MBR

**PSW** Status flags রাখে (Sign, Zero, Carry, Overflow ইত্যাদি)

**Design Considerations** OS support, cost vs speed trade-off

Data registers	
D0	
D1	
D2	
D3	
D4	
D5	
D6	
D7	

Address registers	
A0	
A1	
A2	
A3	
A4	
A5	
A6	
A7	

Program status	
Program counter	
Status register	

(a) MC68000

General registers	
AX	Accumulator
BX	Base
CX	Count
DX	Data

Pointers and index	
SP	Stack ptr
BP	Base ptr
SI	Source index
DI	Dest index

Segment	
CS	Code
DS	Data
SS	Stack
ES	Extrat

Program status	
Flags	
Instr ptr	

(b) 8086

General registers	
EAX	AX
EBX	BX
ECX	CX
EDX	DX

Pointers and index	
ESP	SP
EBP	BP
ESI	SI
EDI	DI

Program status	
FLAGS register	
Instruction pointer	

(c) 80386—Pentium 4

Figure 12.3 Example Microprocessor Register Organizations

## ✿ 12.3 INSTRUCTION CYCLE (নির্দেশনা চক্র)

Processor-এর কাজ হলো বারবার নির্দেশনা (instruction) মেমরি থেকে এনে (fetch করে), সেটি execute করা, এবং মাঝে মাঝে interrupt হ্যান্ডেল করা।  
এই পুরো প্রক্রিয়াটাই বলা হয় — Instruction Cycle।

### ⚙ Instruction Cycle-এর ৩টি প্রধান ধাপ

ধাপ	কাজের বিবরণ
1 Fetch Cycle	পরবর্তী instruction মেমরি থেকে processor-এ আনা হয়।
2 Execute Cycle	Instruction-এর opcode বিশ্লেষণ করে নির্ধারিত কাজ সম্পন্ন করা হয়।
3 Interrupt Cycle	Interrupt ঘটলে বর্তমান কাজের অবস্থা সংরক্ষণ করে interrupt service routine চালানো হয়।

## 🧠 Indirect Cycle (পরোক্ষ চক্র)

যখন কোনো instruction indirect addressing mode ব্যবহার করে, তখন operand-এর আসল address পেতে অতিরিক্ত memory access প্রয়োজন হয়।

→ একে Indirect Cycle বলা হয়, এবং এটি Fetch ও Execute-এর মাঝে ঘটে।

### 🌐 Instruction Cycle-এর ধাপসমূহ (Figure 12.4 অনুসারে):

1. **Fetch:**  
মেমরি থেকে instruction আনা হয়।
2. **Indirect (যদি প্রয়োজন হয়):**  
Instruction যদি indirect addressing ব্যবহার করে, তাহলে আসল operand-এর address বের করতে আরও একবার মেমরি অ্যাক্সেস করা হয়।
3. **Execute:**  
Opcode ডিকোড হয়ে নির্দেশনা অনুযায়ী কাজ (যেমন data move, add, subtract, I/O) করা হয়।
4. **Interrupt (যদি ঘটে):**  
কোনো interrupt ঘটলে বর্তমান PC সংরক্ষণ করে Interrupt Service Routine শুরু হয়।

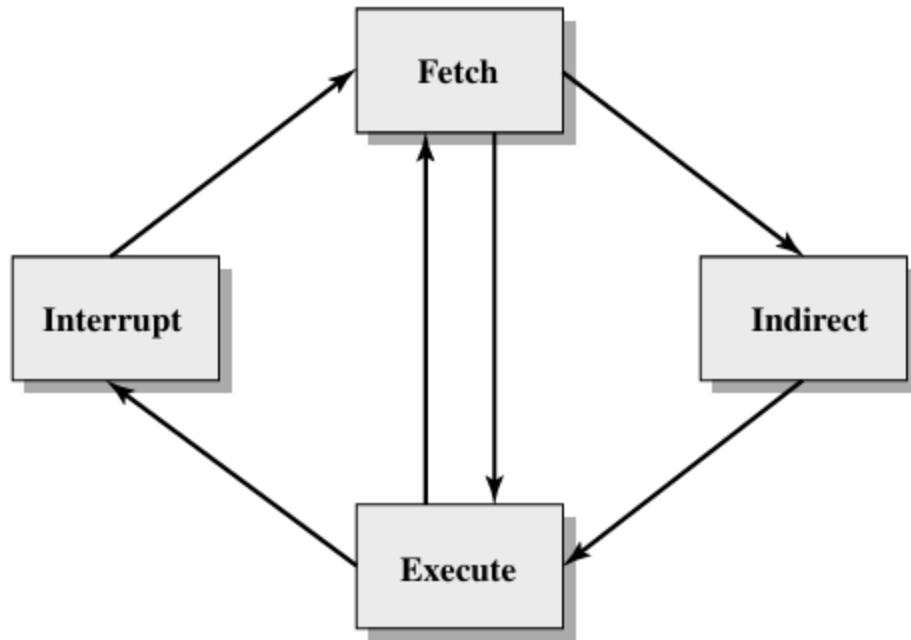


Figure 12.4 The Instruction Cycle



## Data Flow (তথ্য প্রবাহ)

Instruction Cycle-এর প্রতিটি ধাপে কোন রেজিস্টারে কোন ডেটা যায়, সেটা নিচে বোঝানো হলো 📺

### ◆ (1) Fetch Cycle-এর সময় ডেটা প্রবাহ:

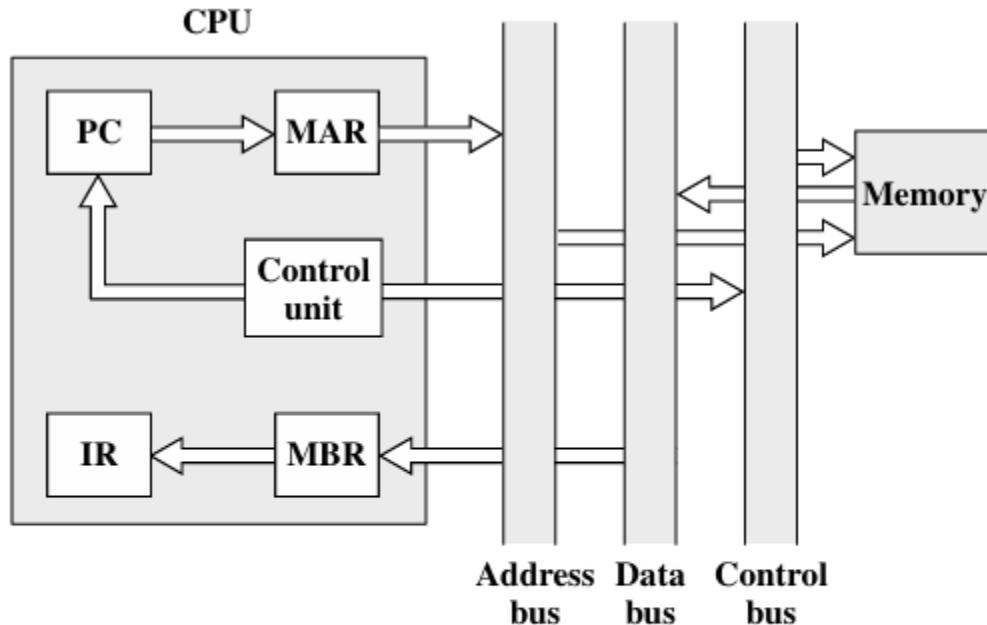
রেজিস্টার জড়িত: PC, MAR, MBR, IR

ধাপগুলো:

ধাপ	কাজ
①	PC → MAR : PC-এর address MAR-এ যায় (যে instruction আনতে হবে)।
②	MAR → Memory Read → MBR : এই address থেকে instruction মেমরি থেকে পড়ে MBR-এ আসে।
③	MBR → IR : Instruction এখন IR-এ রাখা হয়, ডিকোডের জন্য।

ধাপ	কাজ
④	PC + 1 : পরবর্তী instruction-এর address তৈরি হয়।

⊗ **উদ্দেশ্য:** Processor মেমরি থেকে পরবর্তী instruction fetch করে।



MBR = Memory buffer register

MAR = Memory address register

IR = Instruction register

PC = Program counter

Figure 12.6 Data Flow, Fetch Cycle

◆ (2) Indirect Cycle-এর সময় ডেটা প্রবাহ (যদি প্রয়োজন হয়):

রেজিস্টার জড়িত: MAR, MBR

ধাপগুলো:

ধাপ	কাজ
①	MBR-এর ডানদিকের অংশে (address অংশে) থাকা মানটি MAR-এ কপি হয়।
②	Control Unit মেমরি read করে — আসল operand-এর address MBR-এ আসে।

⌚ উদ্দেশ্য: Indirect address resolve করে আসল operand address পাওয়া।

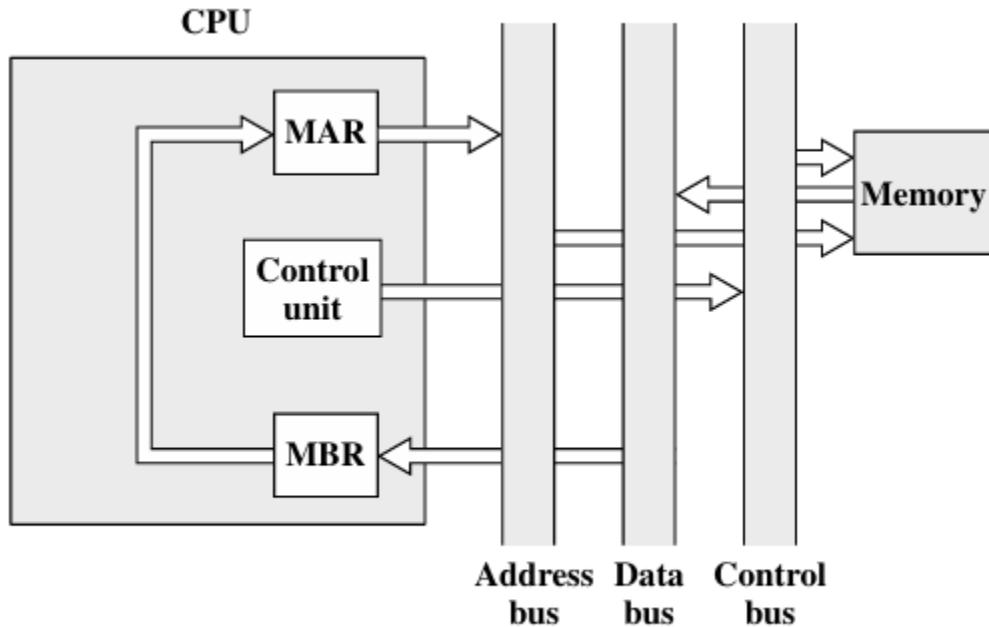


Figure 12.7 Data Flow, Indirect Cycle

---

◆ (3) Execute Cycle-এর সময়:

কাজ নির্ভর করে instruction-এর ধরন অনুযায়ী।  
এখানে হতে পারে:

- Register ↔ Register data transfer
- Memory read/write
- ALU operation (addition, subtraction ইত্যাদি)
- I/O অপারেশন

⌚ উদ্দেশ্য: Instruction-এর নির্ধারিত কাজ সম্পন্ন করা।

---

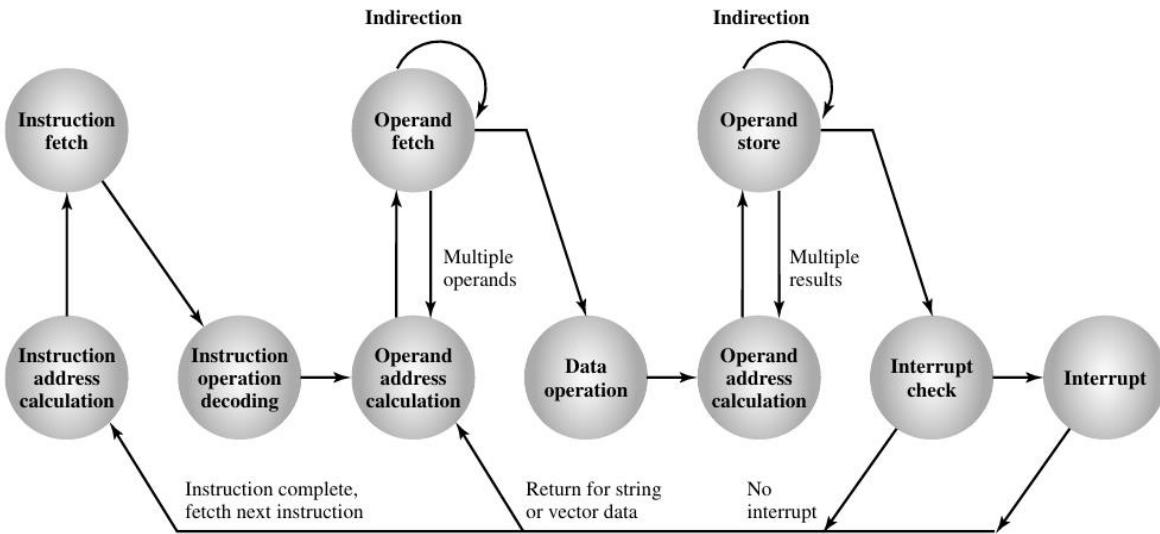


Figure 12.5 Instruction Cycle State Diagram

#### ◆ (4) Interrupt Cycle-এর সময় ডেটা প্রবাহ:

রেজিস্টার জড়িত: PC, MBR, MAR

ধাপগুলো:

ধাপ	কাজের ব্যাখ্যা	কী বোঝায়
①	বর্তমান PC (Program Counter)-এর মান MBR (Memory Buffer Register)-এ কপি করা হয়।	কারণ PC-তে আছে — "প্রবর্তী instruction-এর ঠিকানা", যেটা interrupt শেষ হলে দরকার হবে।
②	Control Unit একটা বিশেষ মেমরি ঠিকানা (interrupt vector বা stack pointer) MAR (Memory Address Register)-এ পাঠায়।	মানে: "এই ঠিকানায় গিয়ে PC-এর মানটা সংরক্ষণ করো"।
③	MBR → Memory — এখন MBR-এ থাকা PC-এর মান মেমরিতে লেখা হয় (যেমন stack-এ বা interrupt table-এ)।	এতে বর্তমান প্রোগ্রামের অবস্থান মেমরিতে সেভ হয়।
④	PC ← ISR Address — এখন PC-তে Interrupt Service Routine (ISR)-এর ঠিকানা লোড হয়।	ফলে processor পরের instruction হিসেবে ISR চালানো শুরু করে।

ঝঃ উদ্দেশ্য: বর্তমান প্রোগ্রামের অবস্থা সংরক্ষণ করে interrupt সার্ভিস শুরু করা।

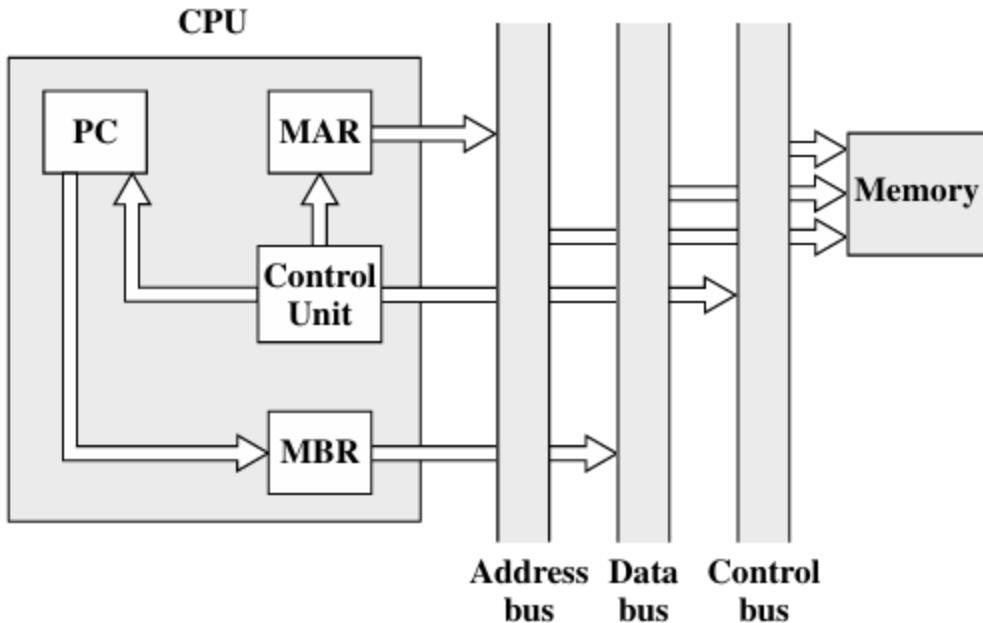


Figure 12.8 Data Flow, Interrupt Cycle

## ■ সংক্ষেপে সারাংশ:

ধাপ	কী ঘটে	সংশ্লিষ্ট রেজিস্টার
Fetch	Instruction মেমরি থেকে আনা হয়	PC, MAR, MBR, IR
Indirect	Operand address resolve হয়	MAR, MBR
Execute	Instruction অনুযায়ী কাজ হয়	ALU, Registers
Interrupt	বর্তমান অবস্থা সংরক্ষণ করে ISR শুরু হয়	PC, MAR, MBR

## ✿ Instruction Pipelining মানে কী?

→ এটা একটা প্রযুক্তি (technique) যার মাধ্যমে processor একই সময়ে একাধিক instruction একসাথে প্রক্রিয়া করে — ঠিক যেমন একটি assembly line-এ অনেক কাজ একসাথে চলে।

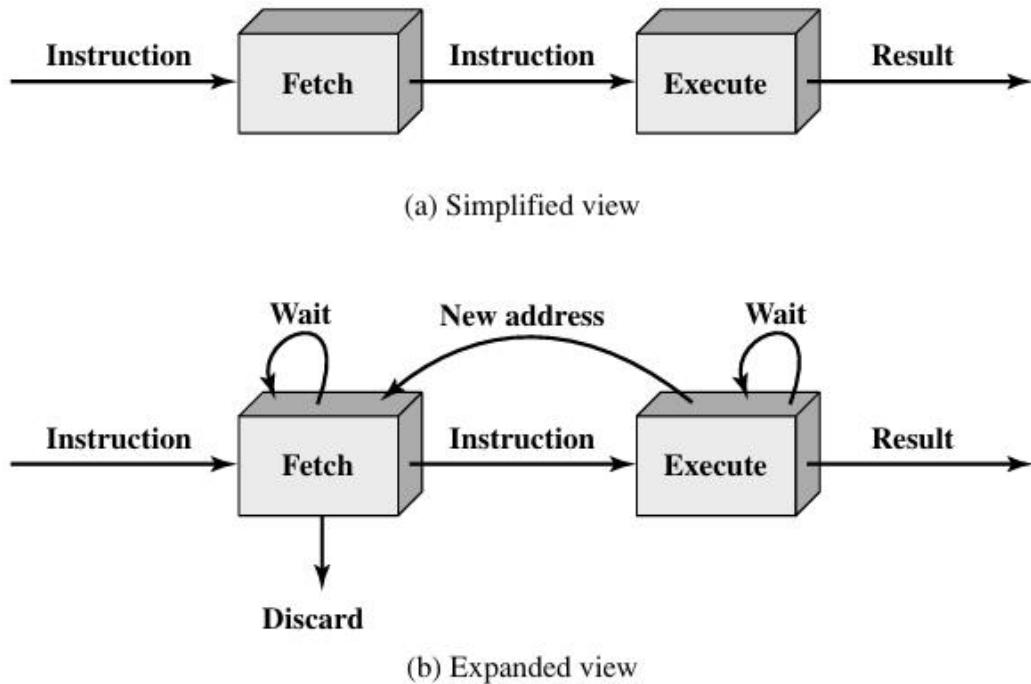


Figure 12.9 Two-Stage Instruction Pipeline



## Assembly Line উদাহরণে বোঝা যাক:

একটা গাড়ি তৈরি করতে ধরো ৪টা ধাপ লাগে:

1. Body তৈরি
2. Paint করা
3. Engine বসানো
4. Test করা

যদি একেকটা গাড়ি এই ৪ ধাপ একটার পর একটা করে বানাও, তবে ১টা গাড়ি শেষ না হলে পরেরটা শুরু করা যাবে না।

কিন্তু যদি তুমি assembly line ব্যবহার করো —

যখন প্রথম গাড়ি paint হচ্ছে, তখনই দ্বিতীয় গাড়ির body তৈরি হচ্ছে, তৃতীয়টার engine বসানো হচ্ছে...

👉 মানে সব ধাপ একসাথে চলছে, ফলে কাজের গতি বেড়ে যায়!

ঠিক এই ধারণাটাই CPU instruction pipelining-এ ব্যবহার হয়।

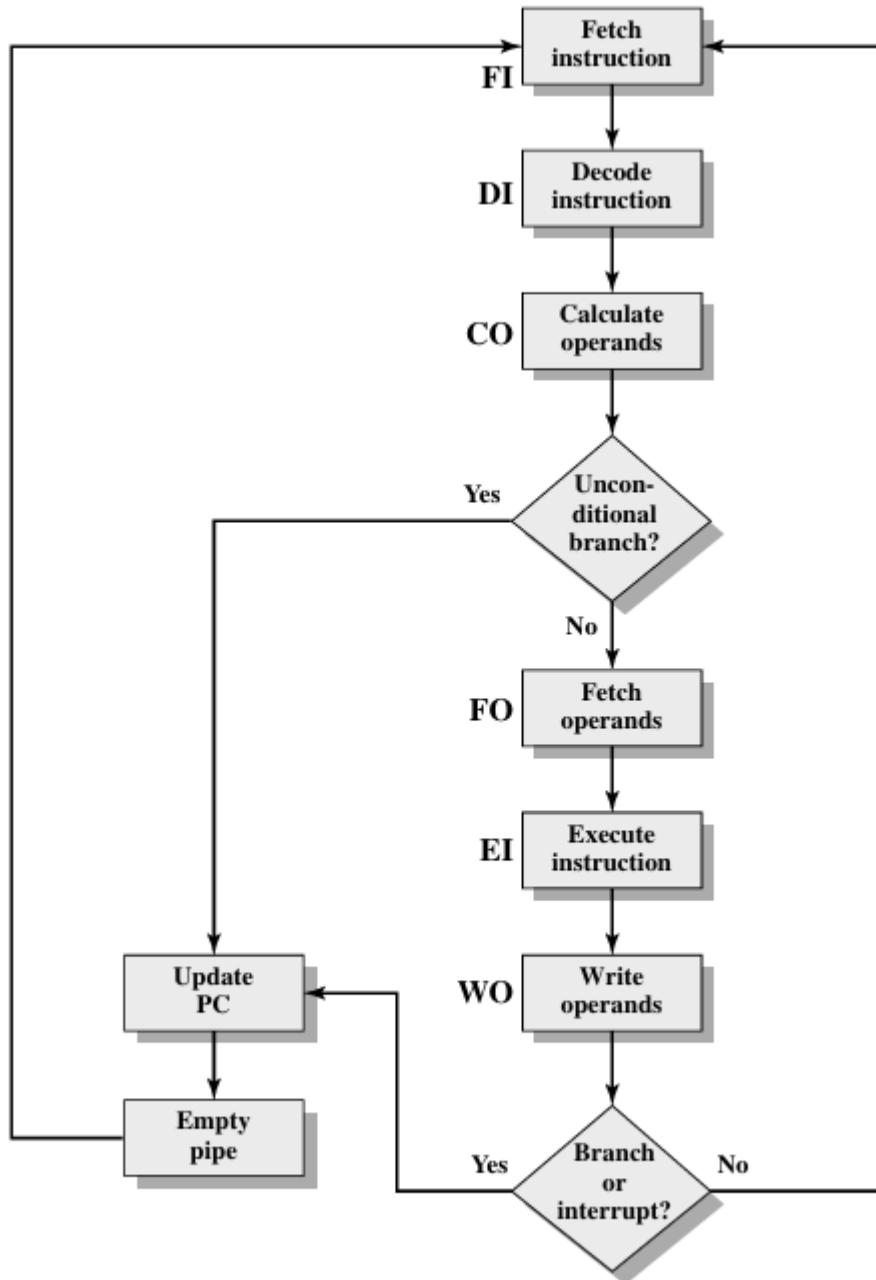


Figure 12.12 Six-Stage CPU Instruction Pipeline



## Instruction Cycle-এর ধাপগুলো মনে করো:

একটা instruction execute করতে সাধারণত এই কাজগুলো হয় 

1. **Fetch Instruction (FI):** Memory থেকে instruction আনা
2. **Decode Instruction (DI):** Opcode/operand চিনে নেওয়া

3. **Calculate Operand (CO):** Address নির্ণয়
  4. **Fetch Operand (FO):** Operand memory থেকে আনা
  5. **Execute Instruction (EI):** কাজ করা
  6. **Write Operand (WO):** Result মেমরিতে লেখা
- 



## Pipeline ধারণা:

ধরো প্রতিটি ধাপ করতে 1 time unit লাগে।

যদি pipelining ছাড়া করা হয় :

👉 9টা instruction লাগবে  $9 \times 6 = 54$  time units

কিন্তু pipelining চালু করলে

👉 এক instruction শেষ হওয়ার আগেই পরেরটা শুরু হয়।

ফলে 9টা instruction শেষ হয় মাত্র 14 time units-এ।

এই ধারণাই বোঝায় **Instruction Pipelining = Parallel Instruction Processing**

---



## উদাহরণ (6-stage pipeline):

	Time →													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

Figure 12.10 Timing Diagram for Instruction Pipeline Operation

👉 6 time unit-এর মধ্যে pipeline full হয়ে যায়।  
তারপর প্রতি time unit-এ 1টা instruction শেষ হচ্ছে।

## ⚠ Pipeline এর সমস্যা (Hazards):

### 1 Structural Hazard:

যখন দুটি stage একসাথে একই hardware (যেমন memory) ব্যবহার করতে চায়।  
→ যেমন: FI আর FO দুটোই memory access করতে চায়।

### 2 Data Hazard:

যখন একটা instruction আগেরটার result-এর উপর নির্ভরশীল।  
→ যেমন:

I1: ADD R1, R2, R3  
I2: SUB R4, R1, R5 ← R1 এখনো I1 থেকে আসেনি!

এতে pipeline থেমে যায় (stall হয়)।

### 3 Control Hazard (Branch Hazard):

যখন branch instruction আসে (if/else, jump ইত্যাদি)।

→ তখন পরের instruction কোনটা হবে, CPU জানে না।

→ Branch নেওয়া হলে আগে fetch করা instruction discard করতে হয় (pipeline flush)।

## ★ Branch Problem-এর উদাহরণ:

	Time →							← Branch penalty						
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO							
Instruction 5					FI	DI	CO							
Instruction 6						FI	DI							
Instruction 7							FI							
Instruction 15								FI	DI	CO	FO	EI	WO	
Instruction 16								FI	DI	CO	FO	EI	WO	

Figure 12.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

ধরো instruction 3 একটা branch —

I3: if (A==0) goto 15

Pipeline meantime I4, I5, I6 already fetch করেছে।

কিন্তু যদি I3 branch নেয়, তাহলে I4–I6 সব ভুল! 😳

তখন এগুলো discard (flush) করতে হয় → সময় নষ্ট হয় (delay)।

## Pipeline Speedup নির্ভর করে:

1. প্রতিটি stage-এর সময় সমান কিনা।
  2. Memory conflict আছে কিনা।
  3. Branch instruction কত ঘনঘন আসে।
  4. Pipeline কত stage-এ ভাগ করা হয়েছে (বেশি stage মানে বেশি parallelism কিন্তু বেশি control logic লাগে)।
- 

## Speedup সূত্র:

$$\text{Speedup} = \frac{\text{Non-pipelined time}}{\text{Pipelined time}}$$

উদাহরণ:

Non-pipelined = 54 units

Pipelined = 14 units

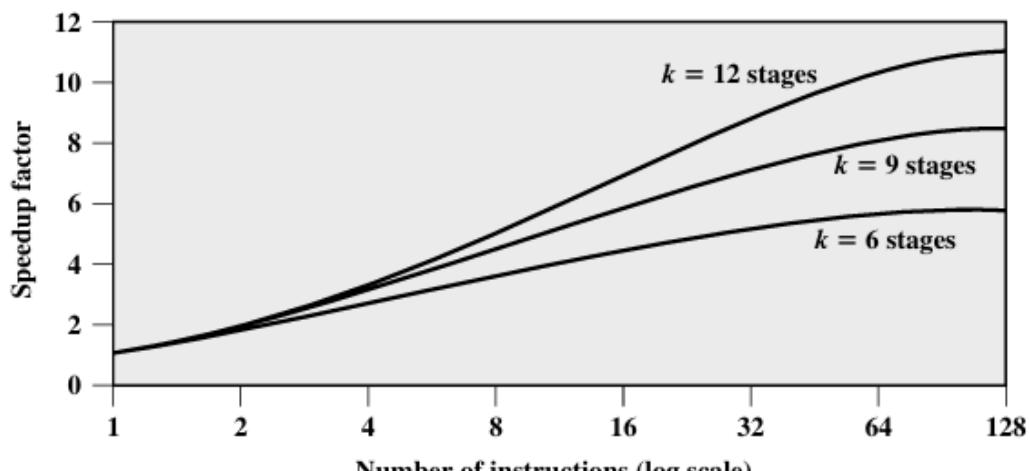
→ Speedup  $\approx 3.85 \times$

## Pipeline-এর Limitations (শেষের অংশের আলোচনা):

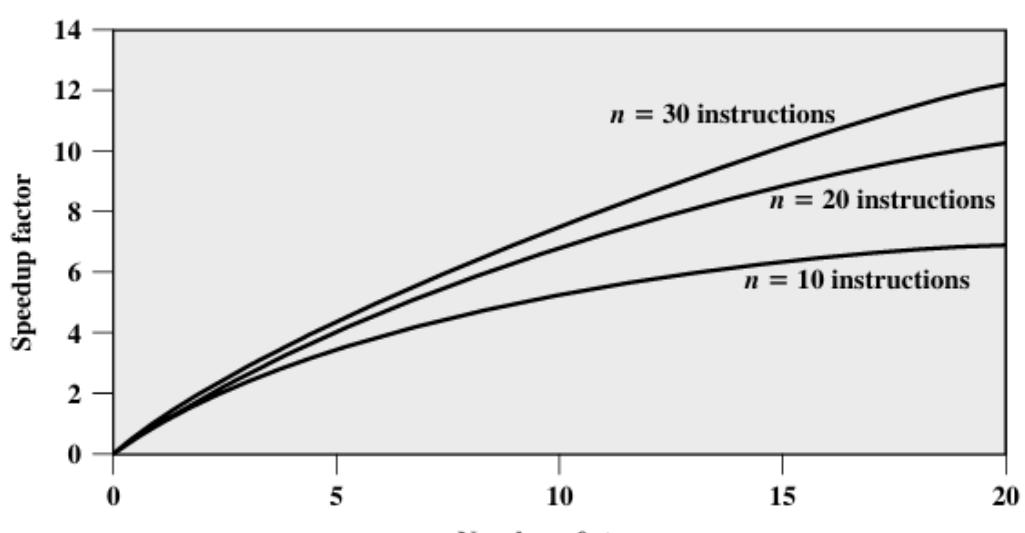
সীমাবদ্ধতা	ব্যাখ্যা
1. Buffer Overhead	প্রতিটি stage-এর মাঝে data move করার জন্য সময় লাগে। এটা instruction latency বাড়ায়।
2. Complex Control Logic	বেশি stage মানে বেশি dependency, hazard detect করার জন্য জটিল সার্কিট দরকার হয়।
3. Latching Delay	Pipeline buffer register-এ data latch করতে সময় লাগে — যা total cycle time বাড়ায়।

## সারসংক্ষেপে:

বিষয়	ব্যাখ্যা
লক্ষ্য	একাধিক instruction একসাথে parallelভাবে process করা
মূল ধারণা	এক instruction-এর এক ধাপ শেষ না হতেই পরের instruction শুরু
ধাপ (ডিটি)	FI, DI, CO, FO, EI, WO
সুবিধা	Execution speed বাড়ে, throughput বাড়ে
সমস্যা	Hazard, branch delay, unequal stage time
সীমাবদ্ধতা	Complex logic, buffer delay, dependency issue



(a)



(b)

Figure 12.14 Speedup Factors with Instruction Pipelining



## 1 Number of Pipeline Stages (k)

মানে — pipeline কত ধাপে (stage) ভাগ করা হয়েছে।

- যত বেশি stage, তত বেশি instruction একসাথে process করা যায় → throughput বেড়ে যায়।
  - কিন্তু বাস্তবে stage বাড়লে কিছু delay (latching delay, hazard, control overhead) বাড়ে, ফলে speedup ধীরে saturate করে।



উপরের গ্রাফে (a):

- যখন  $k = 6, 9, 12$ , দেখা যাচ্ছে —
  - Stage যত বাড়ছে, speedup তত বাড়ছে,
  - কিন্তু একটা সময় পর তা আর বেশি বাড়ছে না (saturation point)।



## 2 Number of Instructions (n)

মানে — প্রোগ্রামে মোট কয়টা instruction আছে যা pipeline দিয়ে যাবে।

- যত বেশি instruction, তত pipeline পূর্ণ (full utilization)।
- ছোট প্রোগ্রামে (কম instruction) pipeline পূর্ণ হওয়ার আগেই শেষ হয়ে যায় → speedup কম হয়।
- বড় প্রোগ্রামে অনেক instruction → steady state আসে → maximum speedup পাওয়া যায়।



নিচের গ্রাফে (b):

- যখন  $n = 10, 20, 30$ , দেখা যাচ্ছে —
  - Instruction সংখ্যা যত বাড়ছে, speedup তত বেশি।
  - কিন্তু একটা সময় পরে gain কমে (কারণ hazards ও overhead বাড়ে)।



## 3 Mathematical Relation:

12  
34

## Mathematical Relation:

Pipeline speedup আনুমানিকভাবে দেওয়া হয়:

$$S = \frac{n \times t_{non-pipe}}{k + (n - 1)} \approx \frac{k}{1 + \frac{k-1}{n}}$$

যেখানে,

- $n$  = total number of instructions
- $k$  = number of pipeline stages

👉 যখন  $n \rightarrow \infty$ , speedup প্রায়  $\approx k$  (number of stages) হয়ে যায়।

## ★ সারসংক্ষেপ:

প্রভাবক	মানে	প্রভাব
Number of Stages ( $k$ )	Pipeline কত ধাপে ভাগ	Stage বাড়লে speedup বাড়ে, কিন্তু বেশি হলে overhead-এর জন্য ধীরে বাড়ে
Number of Instructions ( $n$ )	মোট কত instruction pipeline দিয়ে যাবে	Instruction বেশি হলে pipeline পূর্ণ হয় $\rightarrow$ ভালো speedup
Pipeline Hazard	Dependency, branch, resource conflict	এগুলো speedup কমিয়ে দেয়
Latch Delay & Control Overhead	Stage পরিবর্তনের delay	Ideal speedup থেকে কমে যায়