

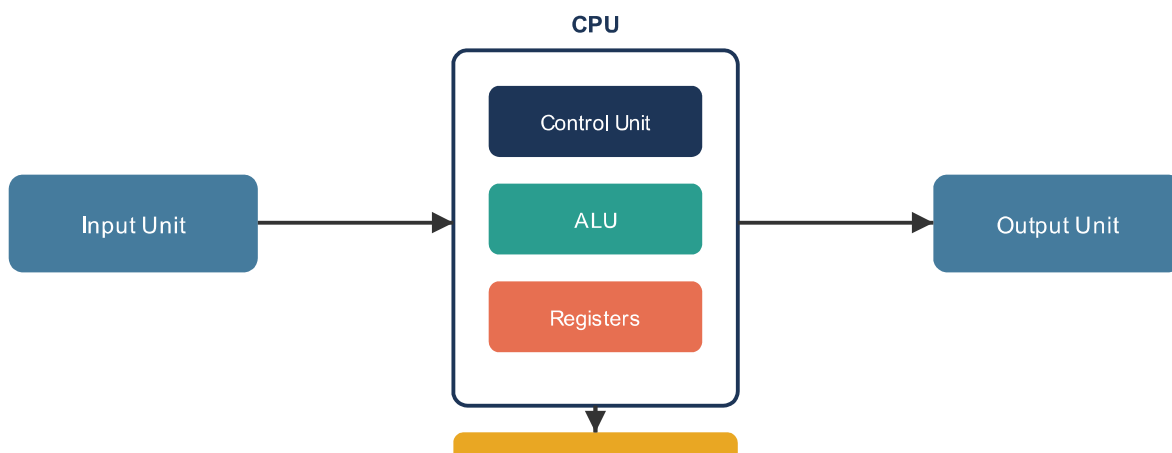
Unit 1: Basic Computer Structure

Functional Blocks, Von Neumann vs Harvard Architecture, Bus Structures, Number System, Registers, Register/Bus/Memory Transfer, RTL, Microoperations, ALU, Data Representation और 2's Complement Arithmetic

1. Computer के Functional Blocks (CPU, Memory, I/O, Control Unit)

कोई भी digital computer मुख्य रूप से चार **functional blocks** से मिलकर बनता है — **Input Unit**, **CPU (Central Processing Unit)**, **Memory Unit** और **Output Unit**। यह सभी blocks आपस में **bus** के through connected होते हैं और मिलकर data को accept, process, store और display करते हैं।

- हर functional block का एक specific role होता है, जिससे पूरा computer system एक coordinated तरीके से काम कर पाता है।
- Data पहले Input Unit से अंदर आता है, फिर CPU उसे process करता है, ज़रूरत पड़ने पर Memory में store/retrieve करता है, और अंत में Output Unit के through result दिखाया जाता है।
- इन सभी blocks को control करने और आपस में synchronize रखने का काम **Control Unit** करता है, जो CPU के अंदर मौजूद होता है।



चित्र 1.1 — Computer के functional blocks: Input Unit से data CPU में जाता है, CPU (Control Unit + ALU + Registers) उसे Memory के साथ मिलकर process करता है और result Output Unit से बाहर आता है।

1.1 Input Unit

- Input Unit वह block है जो बाहरी दुनिया (user) से data और instructions को computer के अंदर लेकर आता है।
- यह human-readable data (जैसे keyboard से typed text) को computer-understandable binary form में convert करता है।
- Examples: Keyboard, Mouse, Scanner, Microphone, Joystick।

1.2 Central Processing Unit (CPU)

CPU को computer का **"brain"** कहा जाता है, क्योंकि यह सभी instructions को fetch, decode और execute करने का काम करता है। CPU मुख्य रूप से तीन parts से मिलकर बनता है।

Part	Function
Control Unit (CU)	Instructions को fetch और decode करता है, और बाकी सभी units को control signals भेजकर उनके काम को coordinate करता है।
Arithmetic and Logic Unit (ALU)	सभी arithmetic (add, subtract, multiply, divide) और logical (AND, OR, NOT, comparison) operations perform करता है।
Registers	छोटी, बहुत तेज़ memory locations होती हैं जो data और instructions को temporarily CPU के अंदर store करती हैं (जैसे AC, PC, IR, MAR, MDR)।

1.3 Memory Unit

- Memory Unit का काम data और instructions को store करना होता है, जब तक उनकी ज़रूरत हो।
- **Primary Memory (Main Memory)** — जैसे RAM और ROM, जो CPU से directly connected होती है और fast access देती है, लेकिन RAM volatile होती है (power off होते ही data delete)।
- **Secondary Memory** — जैसे Hard Disk, SSD, Pen Drive, जो non-volatile होती है और permanently data store करती है, लेकिन speed में primary memory से slower होती है।

1.4 Output Unit

- Output Unit computer द्वारा process किए गए result को binary form से human-readable form में convert करके user को दिखाता है।
- Examples: Monitor, Printer, Speaker, Plotter।

1.5 Control Unit (CU) — विस्तार से

Control Unit CPU का वह भाग है जो instruction cycle के हर step (Fetch, Decode, Execute) को manage करता है और control signals generate करके पूरे computer के सभी components को सही समय पर सही काम करने के लिए direct करता है।

- यह खुद data को process नहीं करता, बल्कि सिर्फ बाकी units को control करता है — इसीलिए इसे "manager" की तरह समझा जा सकता है।
- Control Unit दो तरीकों से design हो सकती है: **Hardwired Control Unit** (fixed logic circuits से बनी) और **Microprogrammed Control Unit** (microinstructions/control memory का use करके बनी)।

Key Point: CPU + Main Memory मिलकर system का "processing core" बनाते हैं, जबकि Input/Output Units बाहरी दुनिया से communication का काम करते हैं। यह पूरा arrangement एक system bus के through आपस में जुड़ा रहता है।

2. Von Neumann Architecture vs Harvard Architecture

Computer architecture यह बताती है कि CPU, Memory और instructions/data के बीच का arrangement कैसा है। इसके दो मुख्य models होते हैं — **Von Neumann Architecture** और **Harvard Architecture**।

2.1 Von Neumann Architecture

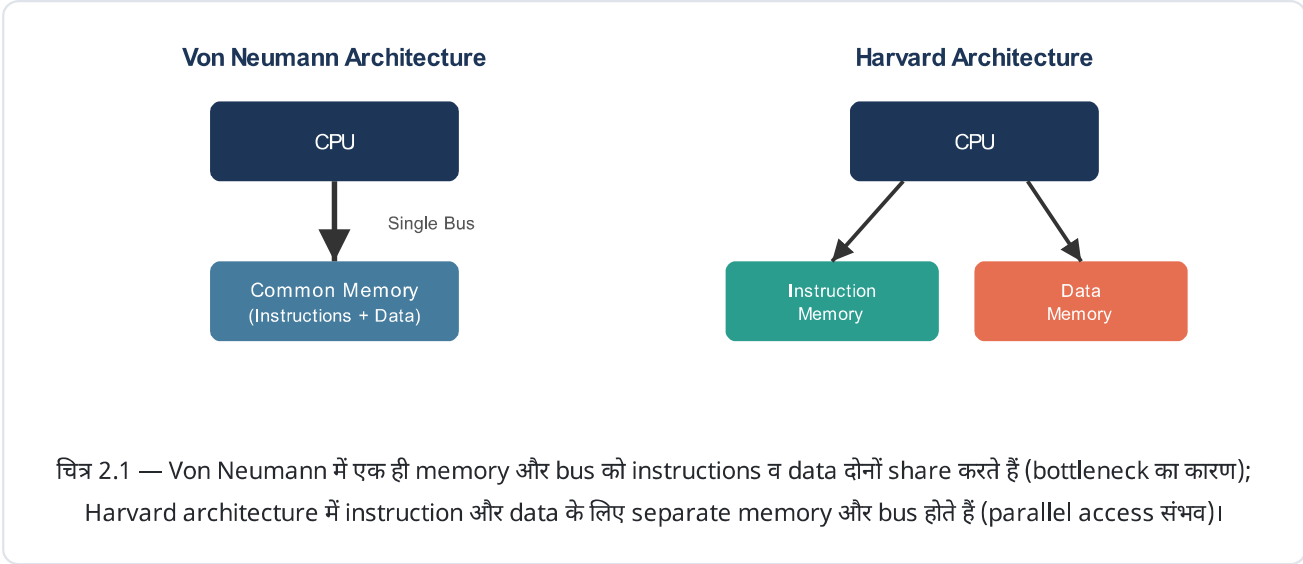
- इस architecture में **instructions और data दोनों एक ही common memory** में store होते हैं, और एक ही bus (system bus) के through access किए जाते हैं।
- CPU एक समय में या तो instruction fetch कर सकता है या data access कर सकता है, दोनों एक साथ नहीं — क्योंकि bus एक ही है।
- Design सरल (simple) है और आज भी अधिकतर general-purpose computers (जैसे PC) इसी architecture पर based होते हैं।

2.2 Von Neumann Bottleneck

Von Neumann Bottleneck: चूँकि instructions और data दोनों को एक ही bus से एक-एक करके transfer करना पड़ता है, इसलिए CPU की processing speed, memory-to-CPU data transfer की speed से limited (bound) हो जाती है। यह limitation "Von Neumann Bottleneck" कहलाती है और यह performance को कम कर सकती है, especially जब CPU बहुत तेज़ हो लेकिन memory access slow हो।

2.3 Harvard Architecture

- इस architecture में **instructions और data के लिए अलग-अलग memory और अलग-अलग buses** होते हैं।
- चूँकि instruction fetch और data access अलग paths से होते हैं, इसलिए CPU दोनों को **parallel (simultaneously)** access कर सकता है, जिससे speed बढ़ती है।
- यह architecture ज़्यादातर **microcontrollers, DSP (Digital Signal Processors)** और embedded systems में use होती है, जहाँ speed critical होती है।



2.4 Comparison Table

Basis	Von Neumann Architecture	Harvard Architecture
Memory	Instructions और Data के लिए एक common memory	Instructions और Data के लिए separate memory
Bus	Single shared bus	Separate buses (parallel access)
Speed	Bottleneck की वजह से comparatively slow	Faster, क्योंकि fetch और access parallel होते हैं

Design Complexity	सरल (Simple)	अधिक complex और costly
Use	General-purpose computers, PCs	Microcontrollers, DSP, embedded systems

Exam Tip: "Von Neumann Bottleneck" एक बहुत common exam question है — याद रखें कि यह single shared bus की वजह से होता है, जहाँ instruction fetch और data transfer एक साथ नहीं हो सकते।

3. Bus Structures — Data, Address और Control Bus

Bus parallel wires/lines का एक समूह (group) होता है, जिसके through CPU, Memory और I/O devices आपस में data, address और control signals को transfer करते हैं। Bus एक **common communication path** की तरह काम करता है, जिसे कई components share करते हैं।

System bus को functionality के according तीन मुख्य types में divide किया जाता है:

3.1 Data Bus

- Data Bus का काम actual data को CPU, Memory और I/O devices के बीच transfer करना होता है।
- यह **bidirectional** होता है — यानी data दोनों directions में (CPU से memory, और memory से CPU) travel कर सकता है।
- Data bus की **width** (जैसे 8-bit, 16-bit, 32-bit, 64-bit) यह decide करती है कि एक समय में कितने bits transfer हो सकते हैं — width जितनी ज़्यादा, speed उतनी बेहतर।

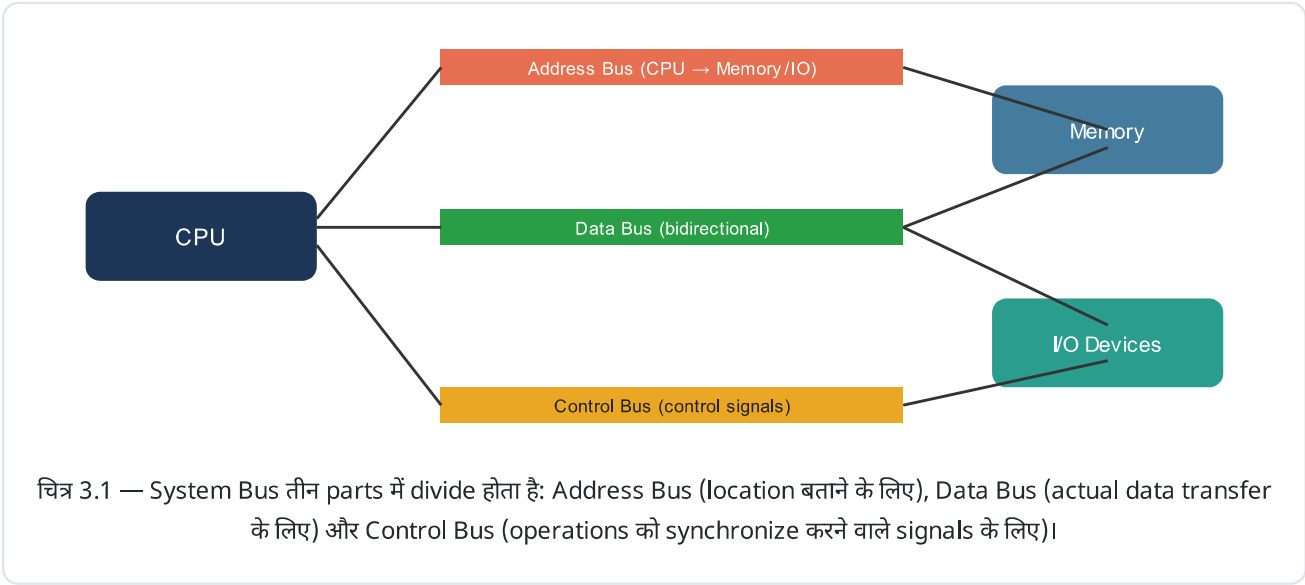
3.2 Address Bus

- Address Bus का काम memory location या I/O device का address बताना होता है, जहाँ से data read करना है या जहाँ data write करना है।
- यह **unidirectional** होता है — address हमेशा CPU से Memory/IO की तरफ ही भेजा जाता है।

- Address bus की width यह decide करती है कि CPU कितनी maximum memory locations को address कर सकता है (n-bit address bus = 2^n addressable locations)।

3.3 Control Bus

- Control Bus का काम control signals (जैसे Read, Write, Memory Select, I/O Select, Interrupt, Clock signals) को carry करना होता है।
- यह Control Unit द्वारा generate किए गए signals को बाकी components तक पहुँचाता है, ताकि सभी operations सही समय पर, सही sequence में हों।
- यह bidirectional हो सकता है, क्योंकि कुछ control signals CPU से बाहर जाते हैं (जैसे Read/Write) और कुछ बाहर से CPU को मिलते हैं (जैसे Interrupt request)।



3.4 Basic Operational Concept

1. Control Unit एक control signal (जैसे "Read") generate करती है और उसे Control Bus पर भेजती है।
2. साथ ही CPU उस memory location का address, जहाँ से data चाहिए, Address Bus पर रखता है।
3. Memory उस address को decode करके requested data को Data Bus पर रख देती है।
4. CPU Data Bus से वह data पढ़ लेता है और उसे register में store कर लेता है।

Bus Type	Direction	Purpose
Data Bus	Bidirectional	Actual data को transfer करना
Address Bus	Unidirectional (CPU → Memory/I/O)	Memory/I/O location का address बताना

Control Bus	Bidirectional	Control signals भेजना (Read, Write, Interrupt आदि)
-------------	---------------	--

4. Number System — Decimal, Binary, Octal, Hexadecimal और Conversion

Number System वह तरीका है जिससे numbers को represent (लिखा) किया जाता है। Computer के अंदर सभी data (चाहे text हो, image हो या instruction) अंततः **binary numbers** के रूप में ही store होता है।

Number System	Base (Radix)	Digits Used
Decimal	10	0–9
Binary	2	0, 1
Octal	8	0–7
Hexadecimal	16	0–9, A–F (A=10, B=11, ... F=15)

4.1 Decimal to Binary/Octal/Hexadecimal Conversion

Decimal number को दूसरे base में convert करने के लिए उसे **बार-बार target base से divide** करते हैं और हर step का remainder नीचे से ऊपर पढ़ते हैं।

```
Example: Decimal 25 को Binary में convert करें
25 / 2 = 12 remainder 1
12 / 2 = 6 remainder 0
6 / 2 = 3 remainder 0
3 / 2 = 1 remainder 1
1 / 2 = 0 remainder 1
```

Answer (नीचे से ऊपर पढ़ें): (25)₁₀ = (11001)₂

4.2 Binary to Decimal Conversion

हर bit को उसके **positional weight (2ⁿ)** से multiply करके सारे results add कर देते हैं।

Example: $(11001)_2$ को Decimal में convert करें

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$= 16 + 8 + 0 + 0 + 1$$

$$= (25)_{10}$$

4.3 Binary to Octal / Octal to Binary

- Binary को Octal में बदलने के लिए bits को दाईं ओर से **3-3 के groups** में बाँटते हैं (क्योंकि $2^3 = 8$)।
- Octal को Binary में बदलने के लिए हर octal digit को उसके **3-bit binary equivalent** से replace कर देते हैं।

Example: $(11001)_2$ को Octal में convert करें

011 001 (3-3 bits के groups, ज़रूरत पड़ने पर आगे 0 add करें)

3 1

Answer: $(11001)_2 = (31)_8$

4.4 Binary to Hexadecimal / Hexadecimal to Binary

- Binary को Hexadecimal में बदलने के लिए bits को दाईं ओर से **4-4 के groups** में बाँटते हैं (क्योंकि $2^4 = 16$)।
- Hexadecimal को Binary में बदलने के लिए हर hex digit को उसके **4-bit binary equivalent** से replace करते हैं।

Example: $(11001)_2$ को Hexadecimal में convert करें

0001 1001

1 9

Answer: $(11001)_2 = (19)_{16}$

Key Point: Octal और Hexadecimal दोनों binary से आसानी से convert होते हैं क्योंकि $8 = 2^3$ और $16 = 2^4$ हैं। इसीलिए Programmers लंबे binary numbers को compact रूप में लिखने के लिए Octal/Hexadecimal का इस्तेमाल करते हैं।

5. Registers, Types of Registers और Instruction Cycle

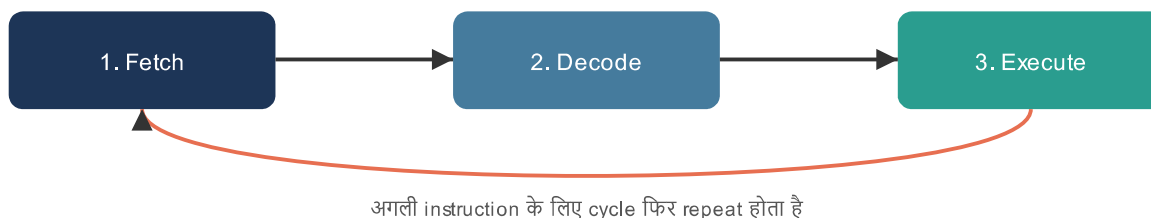
Register CPU के अंदर मौजूद एक बहुत छोटी, बहुत तेज़ (fastest) storage location होती है, जिसका उपयोग data, address या instructions को temporarily store करने के लिए किया जाता है, ताकि CPU उन्हें तुरंत process कर सके। Main memory के comparison में registers की access speed सबसे तेज़ होती है, लेकिन इनकी size बहुत छोटी (कुछ bits/bytes) होती है।

5.1 Types of Registers और उनके Functions

Register	Full Form	Function
PC	Program Counter	अगली execute होने वाली instruction का address रखता है।
IR	Instruction Register	Memory से fetch की गई current instruction को store करता है।
MAR	Memory Address Register	उस memory location का address रखता है, जिसे read/write करना है।
MDR / MBR	Memory Data/Buffer Register	Memory से पढ़ा गया या memory में लिखा जाने वाला data temporarily store करता है।
AC	Accumulator	ALU operations के intermediate results store करता है।
General Purpose Registers	—	Programmer के by आसानी से use होने वाले registers, विभिन्न temporary values store करने के लिए (जैसे R0, R1, R2...)।

5.2 Instruction Cycle (Fetch-Decode-Execute Cycle)

Instruction Cycle वह sequence of steps है, जो CPU एक instruction को memory से लाकर उसे execute करने के लिए बार-बार repeat करता है। यह मुख्यतः तीन steps में होता है — **Fetch, Decode और Execute**।



चित्र 5.1 — Instruction Cycle: Fetch (memory से instruction लाना) → Decode (उसका meaning समझना) → Execute (उसे perform करना), फिर अगली instruction के लिए यही cycle दोबारा चलता है।

1. **Fetch:** PC में मौजूद address का use करके, वह instruction Memory से लाकर IR में store की जाती है, और PC को next instruction के लिए increment कर दिया जाता है।
2. **Decode:** Control Unit उस instruction को decode करके यह पता लगाती है कि कौन-सा operation perform करना है और कौन-से operands चाहिए।
3. **Execute:** ALU या बाकी components के through actual operation perform होता है और result register/memory में store हो जाता है।

याद रखें: हर instruction के लिए यह पूरा cycle (Fetch-Decode-Execute) दोबारा-दोबारा repeat होता है, जब तक program खत्म न हो जाए। कई modern CPUs में एक चौथा step "**Store**" (result को वापस memory में लिखना) भी होता है।

6. Register Transfer, Bus और Memory Transfer, RTL

Computer के अंदर होने वाले सभी operations अंततः registers के बीच data movement और microoperations के रूप में होते हैं। इन operations को symbolically represent करने के लिए **Register Transfer Language (RTL)** का इस्तेमाल किया जाता है।

6.1 Register Transfer

- Register Transfer का मतलब है एक register की content को दूसरे register में copy करना।
- इसे simple arrow notation से represent किया जाता है — जैसे $R2 \leftarrow R1$ का मतलब है $R1$ की value $R2$ में copy हो रही है ($R1$ unchanged रहता है)।

```
R2 ← R1           // R1 का content R2 में transfer होगा
If (P=1) then (R2 ← R1)  // Conditional transfer: सिर्फ तब होगा जब control signal P = 1 हो
```

6.2 Bus Transfer

- जब बहुत सारे registers आपस में connect करने होते हैं, तो हर register के बीच अलग wires लगाना costly हो जाता है — इसलिए एक **common bus** का use किया जाता है, जिसे सभी registers share करते हैं।
- Bus को control करने के लिए **Multiplexers (MUX)** का इस्तेमाल होता है, जो select लाइनों के according decide करते हैं कि किस register का data bus पर रखा जाएगा।
- एक समय पर सिर्फ एक ही register bus पर अपना data भेज सकता है, लेकिन कई registers उसे simultaneously receive कर सकते हैं।

6.3 Memory Transfer

- Memory Transfer का मतलब है Memory और Registers के बीच data का आदान-प्रदान (exchange)।
- इसके लिए दो basic operations होते हैं — **Read** और **Write**।

```
Read Operation:  R1 ← M[AR]
                 (Memory Address Register में दिए गए address से data पढ़कर R1 में store होगा)
```

```
Write Operation: M[AR] ← R1
                 (R1 का content, AR में दिए गए address पर memory में लिखा जाएगा)
```

6.4 Register Transfer Language (RTL)

RTL एक symbolic (shorthand) notation है, जिसका इस्तेमाल digital systems के micro-operations को precisely और compactly describe करने के लिए किया जाता है — बिना विस्तार से plain English में लिखे।

Symbol	Meaning	Example
Letters/Numbers	Register का नाम	MAR, R1, PC, AC
Parentheses ()	Register के अंदर एक specific part/field	PC(L) — PC का low-order byte

Arrow $\leftarrow$	Data transfer की direction	$R2 \leftarrow R1$
Comma ,	एक साथ (simultaneously) होने वाले operations अलग करने के लिए	$R2 \leftarrow R1, R1 \leftarrow R2$ (values swap)
Square brackets []	Memory address को indicate करने के लिए	$M[AR]$

Key Point: RTL का main फायदा यह है कि यह hardware के internal register-level operations को बहुत ही precise और unambiguous तरीके से represent करता है, जिससे digital circuits design करना आसान हो जाता है।

7. Microoperations — Arithmetic, Logical और Shift

Microoperations

Microoperation वह सबसे basic (elementary) operation होती है, जो एक या एक से ज़्यादा registers में store डेटा पर perform की जाती है, और जो सामान्यतः एक single clock pulse में complete हो जाती है। हर machine instruction अंदर से कई microoperations के sequence में execute होती है।

Microoperations को उनके काम के according तीन categories में divide किया जाता है:

7.1 Arithmetic Microoperations

- यह registers में store binary data पर arithmetic operations (जैसे addition, subtraction, increment, decrement) perform करती हैं।

```

R3  $\leftarrow$  R1 + R2      // Addition
R3  $\leftarrow$  R1 - R2      // Subtraction (2's complement से implement होता है)
R1  $\leftarrow$  R1 + 1       // Increment
R1  $\leftarrow$  R1 - 1       // Decrement
R2  $\leftarrow$  ~R2 (Complement)

```

7.2 Logical Microoperations

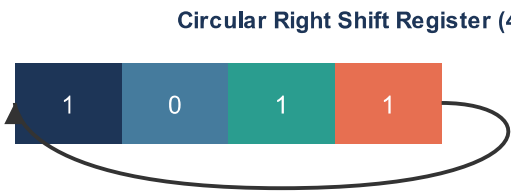
- यह registers के individual bits पर logical operations perform करती हैं, और हर bit independently process होता है (कोई carry involve नहीं होता)।

```
R3 ← R1 AND R2    // Bitwise AND – दोनों bits 1 हों तभी result 1
R3 ← R1 OR R2     // Bitwise OR – कोई भी bit 1 हो तो result 1
R3 ← R1 XOR R2    // Bitwise XOR – bits अलग हों तो result 1
R1 ← NOT R1       // Complement – हर bit invert हो जाता है
```

7.3 Shift Microoperations

- यह register के अंदर bits को left या right की तरफ shift (खिसकाने) का काम करती हैं। यह serial data transfer और multiplication/division (by powers of 2) में उपयोगी होती हैं।

Type	Description
Logical Shift	Bits को left/right shift करता है, और खाली हुई जगह पर 0 insert करता है।
Arithmetic Shift	Signed binary numbers के लिए use होता है — sign bit को preserve रखते हुए shift करता है (right shift में sign bit copy होता है)।
Circular Shift (Rotate)	जो bit एक तरफ से बाहर निकलता है, वही bit दूसरी तरफ वापस insert हो जाता है (कोई bit lose नहीं होता)।



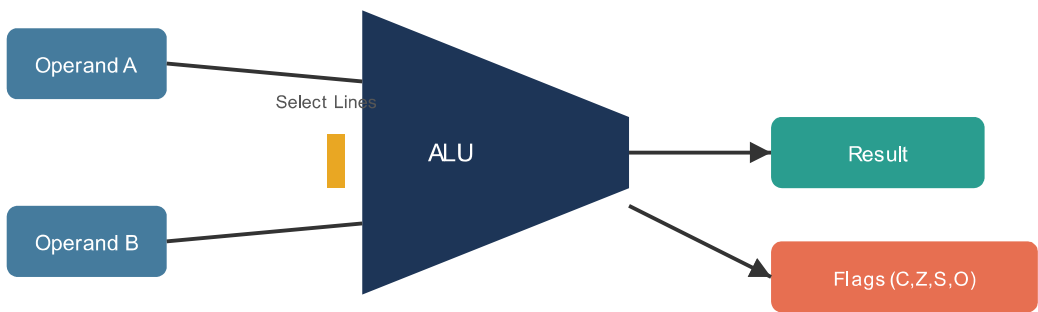
चित्र 7.1 — Circular shift में सबसे right वाला bit बाहर निकलकर सबसे left पर वापस आ जाता है, इसलिए कोई भी bit permanently lose नहीं होता।

Exam Tip: Arithmetic shift और Logical shift में मुख्य फर्क यह है कि Arithmetic shift signed numbers के sign bit को preserve करता है, जबकि Logical shift में हमेशा 0 ही insert होता है — यह एक common comparison question है।

8. ALU और Arithmetic Logic Shift Unit

ALU (Arithmetic and Logic Unit) CPU का वह combinational circuit है, जो दिए गए दो operands पर, select lines के according, arithmetic या logical operation perform करता है और result output करता है।

- ALU दो input operands (जैसे A और B) लेता है, साथ ही कुछ **select/control signals** लेता है जो यह decide करते हैं कि कौन-सा operation perform होगा।
- ALU output के साथ-साथ कुछ **status flags** भी generate करता है, जैसे — Carry, Zero, Sign, Overflow — जो आगे की decisions (जैसे conditional branching) में इस्तेमाल होते हैं।



चित्र 8.1 — ALU दो operands (A, B) और select lines लेकर operation perform करता है, फिर result के साथ Carry, Zero, Sign, Overflow जैसे status flags भी generate करता है।

8.1 Arithmetic Logic Shift Unit (Combined Unit)

कुछ processors में Arithmetic, Logic और Shift — तीनों operations के लिए अलग-अलग circuits बनाने की बजाय, एक ही **combined "Arithmetic Logic Shift Unit"** बना दिया जाता है, जो **select lines** के combination से decide करता है कि कौन-सा operation (Add, Subtract, AND, OR, Shift Left, Shift Right आदि) perform करना है।

Select Lines	Operation Type	Example
00	Arithmetic	Add, Subtract, Increment
01	Logical	AND, OR, XOR, NOT
10	Shift	Shift Left, Shift Right, Rotate

इस design का फायदा यह है कि hardware कम लगता है (एक ही unit से सभी operations हो जाते हैं) और control करना भी आसान हो जाता है, क्योंकि select lines के through ही सही operation choose हो जाता है।

9. Data Representation — Fixed Point और Floating Point

Computer के अंदर numbers (integers और decimal/fractional values दोनों) को store करने के दो main तरीके होते हैं — **Fixed Point Representation** और **Floating Point Representation**।

9.1 Fixed Point Representation of Integers

- Integer numbers को represent करने के लिए एक fixed number of bits use होती हैं, जिनमें से पहला bit (MSB) आमतौर पर **sign bit** होता है (0 = positive, 1 = negative), और बाकी bits magnitude (value) को represent करते हैं।
- Decimal point हमेशा एक fixed position पर माना जाता है (usually सबसे right, इसलिए "fixed point")।

8-bit fixed point representation of +25:

Sign bit: 0 Magnitude: 0011001

=> 0 0011001 (MSB = sign, बाकी 7 bits value)

9.2 Fixed Point Representation of Decimal (Fractional) Numbers

- Fractional numbers (जैसे 5.75) को represent करने के लिए bits को दो parts में divide कर दिया जाता है — **Integer part** और **Fractional part**, जिनके बीच एक assumed (implied) binary point होता है।
- चूँकि binary point की position fixed रहती है, इसलिए इसे "fixed point" representation कहते हैं — लेकिन इसमें बहुत बड़ी range के numbers represent करना मुश्किल होता है।

9.3 Floating Point Representation

Floating Point Representation में numbers को **scientific notation** जैसे रूप में represent किया जाता है: $\text{Number} = \text{Mantissa} \times \text{Base}^{\text{Exponent}}$ । इससे बहुत छोटे और बहुत बड़े

numbers, दोनों को एक ही format में represent किया जा सकता है।

IEEE 754 — Single Precision (32-bit) Format



चित्र 9.1 — IEEE 754 single precision format में कुल 32 bits होते हैं: 1 bit Sign, 8 bits Exponent और 23 bits Mantissa (fractional part)।

- **Sign bit** — number positive है या negative, यह बताता है (0 = positive, 1 = negative)।
- **Exponent** — number की range/magnitude को represent करता है (bias के साथ store होता है)।
- **Mantissa** — number की precision/significant digits को represent करता है।

Basis	Fixed Point	Floating Point
Range	सीमित (Limited) range	बहुत बड़ी range (very large/small numbers)
Precision	कम bits की वजह से limited	ज़्यादा precise (mantissa की वजह से)
Complexity	Implement करना आसान	Implement करना ज़्यादा complex
Speed	Fast operations	Comparatively slower operations
Use	Simple integer calculations	Scientific calculations, real numbers

10. Negative Number Representation — Sign Bit, 1's Complement, 2's Complement

Binary system में negative numbers को represent करने के मुख्यतः तीन तरीके होते हैं — **Sign-Magnitude**, **1's Complement** और **2's Complement**। इनमें से 2's Complement सबसे ज़्यादा use होने वाला method है, क्योंकि यह arithmetic operations को सबसे simple बनाता है।

10.1 Sign-Magnitude Representation (Using Sign Bit)

- सबसे पहला bit (MSB) sign बताता है — 0 = positive, 1 = negative — और बाकी bits सिर्फ number की magnitude (absolute value) बताते हैं।

+5 (8-bit) = 0 0000101

-5 (8-bit) = 1 0000101 (सिर्फ sign bit बदला, magnitude same)

Problem: इसमें **दो zeros** बन जाते हैं — +0 (00000000) और -0 (10000000), जिससे arithmetic circuits बनाना complex हो जाता है।

10.2 1's Complement Representation

- Negative number बनाने के लिए positive number के **हर bit को invert (0→1, 1→0)** कर दिया जाता है।

+5 (8-bit) = 0000 0101

-5 (1's Complement) = हर bit invert करें = 1111 1010

Problem: इसमें भी **दो zeros** (+0 और -0) बनते हैं, और addition में एक extra step ("end-around carry") add करना पड़ता है।

10.3 2's Complement Representation

2's Complement निकालने के लिए, पहले number का 1's complement लेते हैं, फिर उसमें **1 add** कर देते हैं। यह method आज लगभग सभी modern computers में negative numbers store करने के लिए use होता है।

+5 (8-bit) = 0000 0101

1's Complement = 1111 1010

+1 + 1

2's Complement of 5 = 1111 1011 (यही -5 को represent करता है)

10.4 Comparison Table

Basis	Sign-Magnitude	1's Complement	2's Complement
-------	----------------	----------------	----------------

Negative बनाने का तरीका	सिर्फ sign bit बदलना	सभी bits invert करना	1's complement + 1
Zero Representation	दो zeros (+0, -0)	दो zeros (+0, -0)	सिर्फ एक unique zero
Addition/Subtraction Circuit	Complex (sign अलग से handle)	Extra end-around carry चाहिए	सबसे simple — एक ही adder circuit से add/subtract दोनों
Use में	बहुत कम	बहुत कम	Modern computers में standard

Key Point (Exam Important): 2's Complement सबसे ज़्यादा use होता है क्योंकि इसमें सिर्फ एक zero होता है और subtraction को भी addition के रूप में ही perform किया जा सकता है — इससे hardware design बहुत सरल हो जाता है।

11. 2's Complement से Arithmetic Addition और Subtraction

2's Complement representation का सबसे बड़ा फायदा यह है कि इससे **subtraction को भी addition की तरह ही** perform किया जा सकता है — बस subtrahend (जिसे घटाना है) का 2's complement लेकर, उसे minuend में add कर दिया जाता है।

11.1 Addition using 2's Complement

- दोनों numbers (positive या negative, जो भी हों) को उनके 2's complement form में सीधे add कर दिया जाता है।
- अगर addition के बाद एक extra carry-out (MSB से बाहर) आता है, तो उसे **discard (ignore)** कर दिया जाता है (जब तक overflow की condition न हो)।

Example: (+5) + (-3) को 2's complement से add करें (8-bit)

```
+5  = 0000 0101
-3  = 1111 1101  (2's complement of 3)
-----
```

Sum = 1 0000 0010 → carry out को discard करें
 Result = 0000 0010 = +2 ✓ (Correct!)

11.2 Subtraction using 2's Complement

- Subtraction ($A - B$) को perform करने के लिए, B का 2's complement निकालकर उसे A में add कर दिया जाता है: $A - B = A + (2\text{'s complement of } B)$

Example: $7 - 5$ को 2's complement से solve करें (8-bit)

7 = 0000 0111

2's comp of 5 = 1111 1011

Sum = 1 0000 0010 → carry out discard करें

Result = 0000 0010 = 2 ✓ ($7 - 5 = 2$, Correct!)

11.3 Overflow Condition

Overflow तब होता है जब result इतना बड़ा हो जाता है कि वह दिए गए bits की range में fit नहीं होता (जैसे दो positive numbers add करने पर negative result आ जाए, या दो negative numbers add करने पर positive result आ जाए)। इसे पहचानने का rule है: **अगर sign bit की तरफ आने वाला carry (carry-in) और सबसे आखिरी carry (carry-out) अलग-अलग हों, तो overflow हुआ है।**

2's Complement Method के Advantages

- Addition और Subtraction दोनों के लिए सिर्फ एक ही adder circuit चाहिए — subtraction के लिए अलग circuit नहीं बनाना पड़ता।
- सिर्फ एक unique zero होता है, जिससे comparison operations आसान हो जाते हैं।
- Hardware design simple और cost-effective हो जाता है।

ध्यान देने योग्य बातें (Limitations)

- Overflow की सही पहचान करने के लिए extra logic circuit चाहिए होता है।
- Fixed bit-width की वजह से representable numbers की range सीमित (limited) होती है।

Exam Tip: 2's Complement addition/subtraction के numerical questions exams में बहुत common हैं — हमेशा carry-out को discard करना याद रखें (जब तक overflow न हो), और final answer को वापस decimal में convert करके verify ज़रूर करें।