**1. Block Level Description of the Functional Units**

A computer is made up of several **functional blocks**, each responsible for specific tasks.

1. **Input Unit**
   * Accepts data and instructions from the user.
   * Converts input into a form the computer can understand (binary).
   * Example: A keyboard sends ASCII codes of typed characters, a scanner converts a photo into digital signals.
2. **Output Unit**
   * Converts computer results into a human-readable form.
   * Example: A monitor displays text/graphics, a printer provides a hard copy, a speaker outputs audio.
3. **Memory (Storage) Unit**
   * Used to **store instructions, input data, intermediate results, and final results**.
   * Types:
     + **RAM (Random Access Memory):** Temporary storage, erased when power goes off.
     + **ROM (Read-Only Memory):** Permanent storage, contains startup programs (BIOS).
   * Example: While playing a video, RAM stores frames temporarily, while ROM holds boot programs.
4. **Central Processing Unit (CPU)**
   * Called the *brain* of the computer.
   * **Control Unit (CU):** Directs flow of instructions and controls input/output operations.
   * **Arithmetic Logic Unit (ALU):** Performs calculations (ADD, SUB) and logic operations (AND, OR).
   * **Registers:** Small, fast memory locations for temporary storage.
   * Example: In ADD AX, BX, ALU adds numbers, CU coordinates the operation, and result is stored in AX.

👉 **Analogy**: Think of a computer as a school:

* Input = Students entering.
* Memory = Library storing books.
* CPU (CU + ALU) = Teacher explaining and solving problems.
* Output = Students giving answers in exams.

**2. Von Neumann Model**

The **Von Neumann Architecture**, proposed in 1945, is the most common computer design today.

1. **Stored Program Concept**
   * Both instructions and data are stored in the **same memory**.
   * This makes the design simple and flexible.
2. **Main Components**
   * **Arithmetic Logic Unit (ALU):** Does math and logic.
   * **Control Unit (CU):** Directs fetching, decoding, and execution.
   * **Memory:** Stores instructions and data.
   * **Input/Output:** Handles external communication.
   * **System Bus:** Connects CPU, memory, and I/O (Data Bus, Address Bus, Control Bus).
3. **Fetch–Decode–Execute Cycle**
   * **Fetch:** CPU takes instruction from memory using Program Counter (PC).
   * **Decode:** CU interprets the instruction.
   * **Execute:** ALU carries out the operation.
   * **Store:** Result saved in memory/register.
4. **Advantages**
   * Simple, flexible, low cost.
5. **Limitations**
   * **Von Neumann Bottleneck:** One bus is used for both data and instructions → CPU may sit idle while waiting.
   * Risk of overwriting instructions with data.

👉 **Example**: In a calculator app, CPU fetches ADD instruction, fetches numbers 5 and 7, executes addition, and stores result = 12.

**3. Computer Organization vs Computer Architecture**

1. **Computer Architecture**
   * Focuses on the **logical structure** of the computer as seen by programmers.
   * Includes: Instruction set, addressing modes, word length, and I/O mechanisms.
   * Example: In x86, the instruction MOV AX, BX is part of the architecture.
2. **Computer Organization**
   * Deals with **how architecture is implemented in hardware**.
   * Includes: Control signals, ALU design, cache size, pipeline depth, memory type.
   * Example: Intel and AMD both follow x86 architecture but use different cache organizations.
3. **Key Difference**
   * Architecture = “What a computer can do.”
   * Organization = “How it actually does it.”
4. **Analogy**
   * Architecture is like the **blueprint of a building** (design).
   * Organization is like the **actual construction** (materials, wiring, layout).

**4. Performance Measures of Computer Architecture**

Performance tells us how **fast and efficient** a computer is.

1. **Response Time (Latency):**
   * Time taken to finish one task.
   * Includes CPU execution, I/O delays, OS overhead.
   * Example: How long it takes for a webpage to load.
2. **Execution Time (CPU Time):**
   * Time CPU spends on a program.
   * Formula:  
     **CPU Time = Instruction Count × CPI × Clock Cycle Time**
   * Example: A program with 15,000 instructions, CPI=12, Clock=3.3 GHz → CPU Time = 54 μs.
3. **Throughput:**
   * Amount of work done per second (tasks completed).
   * Example: A web server handles 2000 requests per second.
4. **Other Factors:**
   * Processor speed (GHz), memory speed (RAM, cache), I/O performance, and software optimization.

👉 **Analogy**: A restaurant’s speed =

* Response Time = How long one customer waits for food.
* Execution Time = How long chef cooks the dish.
* Throughput = How many customers served in 1 hour.

**5. Von Neumann Architecture + Performance Measures**

1. **Architecture Features:**
   * Uses **one memory** for both instructions and data.
   * Works on **Fetch–Decode–Execute cycle**.
   * Components: CPU, Memory, Input/Output, System Bus.
2. **Advantages:**
   * Simpler design, flexible for running any program, cost-effective.
3. **Limitations:**
   * **Bottleneck**: CPU and memory share same bus.
   * Only one instruction executed at a time.
4. **Performance Metrics:**
   * **Execution Time, Response Time, Throughput.**
   * Example: SPEC benchmark measures real program speed.

**6. 8086 Microprocessor Architecture**

The Intel 8086 is a **16-bit microprocessor** that introduced advanced features compared to earlier processors.

1. **Basic Features:**
   * 20-bit address bus → can access 1 MB memory.
   * 16-bit data bus → can read/write 16 bits at a time.
   * Supports **pipelining** using two functional units.
2. **Bus Interface Unit (BIU):**
   * Generates physical addresses using segment + offset.
   * Prefetches up to 6 instructions into a queue.
   * Manages memory and I/O transfers.
3. **Execution Unit (EU):**
   * Decodes instructions from BIU queue.
   * Executes using ALU and registers.
   * Updates Flag Register after execution.
4. **Advantage – Pipelining:**
   * While EU executes, BIU fetches new instructions.
   * Increases speed compared to older processors like 8085.

**7. Instruction Set of 8086**

The 8086 instruction set is divided into groups:

1. **Data Transfer Instructions**
   * Move data between registers, memory, or I/O.
   * Example: MOV AX, BX, PUSH CX.
2. **Arithmetic Instructions**
   * Perform math operations.
   * Example: ADD AX, BX, MUL CX.
3. **Logical Instructions**
   * Perform bitwise logic.
   * Example: AND AL, 0Fh, OR AX, BX.
4. **Control Transfer Instructions**
   * Change program flow.
   * Example: JMP LABEL, CALL PROC, RET.
5. **String Instructions**
   * Operate on memory blocks.
   * Example: MOVSB (move byte string), CMPSB (compare strings).

**8. Addressing Modes of 8086 (with Examples)**

Addressing modes tell the CPU **where the operand is located**.

1. **Immediate Addressing**
   * Operand is given directly in instruction.
   * Example: MOV AX, 25H.
2. **Register Addressing**
   * Operand stored in a register.
   * Example: MOV AX, BX.
3. **Direct Addressing**
   * Operand located at specific memory address.
   * Example: MOV AX, [2000H].
4. **Register Indirect Addressing**
   * Address stored in register, operand fetched from memory.
   * Example: MOV AX, [BX].
5. **Base-Indexed Addressing**
   * Effective address = Base register + Index register.
   * Example: MOV AX, [BX+SI].

👉 **Analogy**:

* Immediate = value written directly.
* Register = data already in CPU’s hand.
* Direct = go to a fixed cupboard (address).
* Register indirect = go to cupboard number written in a register.
* Base-indexed = cupboard = base number + index number.

Got it 👍 From **Q9 onward**, I’ll **expand further**, explain in a **simpler and easy-to-understand way**, and include **examples** where possible. These answers will be like **exam notes but in a student-friendly style**.

**9. Features of 8086 Microprocessor**

The **Intel 8086** was the first 16-bit processor and is the base of the x86 family. Its important features are:

1. **Word Size & Memory Access**
   * It is a **16-bit microprocessor**.
   * It can process 16 bits of data in one operation (so faster than 8-bit processors like 8085).
   * Has a **20-bit address bus**, meaning it can access up to **1 MB (2²⁰ = 1,048,576 bytes)** of memory.
2. **Memory Segmentation**
   * Memory is divided into segments: **Code Segment, Data Segment, Stack Segment, Extra Segment**.
   * Each segment is of size **64 KB**.
   * Segmentation allows **better memory management** and supports **modular programming**.
   * Example: One program can keep instructions in the Code Segment and variables in the Data Segment.
3. **Pipelining Support**
   * 8086 has **two units**: Bus Interface Unit (BIU) and Execution Unit (EU).
   * BIU fetches instructions in advance and stores them in a **6-byte instruction queue**.
   * EU executes instructions from the queue.
   * This overlap is called **pipelining**, which makes execution faster.
4. **Other Features**
   * Has **14 registers** (AX, BX, CX, DX, CS, DS, SS, ES, SP, BP, SI, DI, IP, Flags).
   * Supports **64K I/O ports** for device communication.
   * Operates in **two modes**:
     + **Minimum mode** – for single processor systems.
     + **Maximum mode** – for multiprocessor systems.
   * Clock frequency ranges between **5 MHz to 10 MHz**.

👉 **Simple Analogy**: Think of 8086 like a fast office worker with separate folders (segments) to keep work organized, and an assistant (BIU) who brings new work while the worker (EU) finishes current tasks.

**10. Bus Interface Unit (BIU)**

The **BIU** acts like the “postman” of the processor – it handles all communication with **memory** and **I/O devices**.

1. **Address Generation**
   * BIU generates **20-bit physical addresses**.
   * Formula:  
     **Physical Address = Segment Address × 10H + Offset**.
   * Example: If CS = 1000H, IP = 2345H → PA = 1000 × 10H + 2345 = 12345H.
2. **Instruction Queue (Prefetching)**
   * BIU fetches up to **6 instruction bytes** in advance and stores them in a queue.
   * This makes execution faster because EU doesn’t have to wait for instructions.
   * Example: While EU is executing ADD AX, BX, BIU is already fetching the next instruction.
3. **Bus Operations**
   * Handles all external memory and I/O transfers.
   * Performs **read and write operations** for data and instructions.
   * Example: Reading input from a keyboard port or writing output to display memory.
4. **Components of BIU**
   * **Segment Registers**: CS, DS, SS, ES.
   * **Instruction Pointer (IP)** – holds the offset of next instruction.
   * **Instruction Queue** – stores prefetched instructions.

👉 **Simple Analogy**: BIU is like a librarian – it fetches books (instructions/data) from shelves (memory) and hands them to the reader (EU).

**11. Register Organization in 8086**

Registers are **small, fast storage areas** inside the CPU used during execution. The 8086 has **14 programmer-accessible registers**, divided into groups:

1. **General Purpose Registers (AX, BX, CX, DX)**
   * 16-bit registers, but each can be split into two 8-bit registers (AH & AL, BH & BL, etc.).
   * **AX (Accumulator):** Used in arithmetic, logic, and I/O operations. Example: ADD AX, 05H.
   * **BX (Base Register):** Often used to form memory addresses. Example: [BX] refers to memory at address stored in BX.
   * **CX (Counter):** Used as a loop counter. Example: LOOP LABEL decreases CX until it becomes 0.
   * **DX (Data Register):** Used in multiplication/division and I/O port addressing. Example: Port address in DX for IN AX, DX.
2. **Segment Registers (CS, DS, SS, ES)**
   * **CS (Code Segment):** Points to code (instructions).
   * **DS (Data Segment):** Points to program data.
   * **SS (Stack Segment):** Points to stack memory (used in function calls, temporary storage).
   * **ES (Extra Segment):** Used in string operations.
3. **Pointer and Index Registers (SP, BP, SI, DI)**
   * **SP (Stack Pointer):** Points to top of stack in SS.
   * **BP (Base Pointer):** Used to access parameters in stack.
   * **SI (Source Index):** Used in string operations (source).
   * **DI (Destination Index):** Used in string operations (destination).
4. **Instruction Pointer (IP) & Flag Register**
   * **IP:** Stores offset of the next instruction to execute.
   * **Flags:** Store CPU status (e.g., ZF = 1 if result is 0).

👉 **Simple Analogy**: Registers are like “drawers on a desk” – very close and quick to use compared to going to a cabinet (RAM).

**12. Execution Unit (EU)**

The **EU** is the “worker” of the processor – it executes instructions fetched by BIU.

1. **Instruction Decoding**
   * EU takes instructions from BIU’s queue.
   * It **decodes** the instruction (understands what operation to do).
2. **Execution using ALU**
   * Performs arithmetic operations (ADD, SUB, MUL, DIV).
   * Performs logical operations (AND, OR, NOT, CMP).
   * Example: For ADD AX, BX, EU adds BX to AX.
3. **Control Circuitry**
   * Directs internal operations of EU.
   * Handles sequencing of instruction execution.
4. **Flags Update**
   * After execution, EU updates **Flag Register**.
   * Example: After subtraction, if result = 0 → Zero Flag (ZF) is set.

👉 **Simple Analogy**: EU is like a chef in a restaurant – BIU brings ingredients (data & instructions), EU cooks (executes), and updates flags (status of the dish).

**13. Flag Register of 8086**

The **Flag Register** is a **16-bit register** used to show CPU status and control operations. It has **9 active flags**, divided into two types:

1. **Status Flags (Result-related)**
   * **CF (Carry Flag):** Set if there is a carry/borrow. Example: FFH + 01H → CF=1.
   * **ZF (Zero Flag):** Set if result = 0. Example: SUB AX, AX → ZF=1.
   * **SF (Sign Flag):** Set if result is negative (MSB = 1).
   * **OF (Overflow Flag):** Set if result too large for signed number. Example: 127 + 1 → OF=1.
   * **PF (Parity Flag):** Set if result has even number of 1s.
   * **AF (Auxiliary Carry):** Set if carry occurs from lower nibble (used in BCD arithmetic).
2. **Control Flags (User-controlled)**
   * **IF (Interrupt Flag):** Enables/disables hardware interrupts. Example: IF=0 disables interrupts.
   * **DF (Direction Flag):** Controls string operations. DF=0 → forward, DF=1 → backward.
   * **TF (Trap Flag):** Enables single-step debugging.
3. **How Flags Affect Instructions**
   * Many instructions check flags for decision-making.
   * Example: JZ LABEL → jumps if Zero Flag = 1.
4. **Importance**
   * Flags help in program control (conditional jumps).
   * They reflect the current status of CPU operations.

👉 **Simple Analogy**: Flags are like “warning lights” in a car dashboard – they tell the current status (empty fuel, engine overheating) and affect decisions (when to stop or refuel).

**1. Instruction Formats**

Instruction formats describe how the bits of an instruction are arranged. Each instruction is made up of multiple fields that tell the CPU what operation to perform and on which data.

* **Opcode Field**: Specifies the operation (like ADD or SUB).
* **Address Field**: Contains the location (memory or register) of the operand.
* **Mode Field**: Specifies how to interpret the operands (e.g., direct or indirect addressing).

Instruction lengths vary depending on the number of fields and their sizes. Longer instructions allow addressing more memory but take more time to fetch.

Instruction formats are mainly categorized based on the number of addresses specified:

* **Zero-address instructions**: Operate implicitly on a stack without explicitly specifying operands.
* **One-address instructions**: Use an accumulator and specify one operand address.
* **Two-address instructions**: Specify two operands where one may be both source and destination.
* **Three-address instructions**: Specify three operands for full flexibility.

For example, in a one-address instruction like ADD X, the CPU adds the value at memory location X to the accumulator.COA-Module-02-1.pptx

**2. Basic Instruction Cycle with Interrupt Processing**

The instruction cycle is the process the CPU follows to execute each instruction, typically involving these steps:

1. **Fetch**: The CPU fetches the instruction from memory using the program counter (PC).
2. **Decode**: The CPU decodes the fetched instruction to understand what operation is needed.
3. **Operand Fetch**: If the instruction needs data, the CPU fetches operands from memory.
4. **Execute**: The CPU performs the operation (e.g., add, subtract).
5. **Interrupt Processing**: If an interrupt occurs (external event requiring attention), the CPU suspends the current process, saves the PC, and jumps to the interrupt service routine. After handling, it resumes the main program.

The cycle repeats for every instruction sequentially unless interrupted.COA-Module-02-1.pptx

**3. Program to Evaluate X=A[B+C(D+E)]F(G+H)X = \frac{A[B + C(D + E)]}{F(G + H)}X=F(G+H)A[B+C(D+E)] using One-Address Instructions**

Using one-address instructions (where the accumulator is used), the program would look like this:

text

LOAD H ; AC <- M[H]

ADD G ; AC <- AC + M[G]

MUL F ; AC <- AC \* M[F]

STORE T ; Store intermediate result in T

LOAD D ; AC <- M[D]

ADD E ; AC <- AC + M[E]

ADD B ; AC <- AC + M[B]

MUL A ; AC <- AC \* M[A]

DIV T ; AC <- AC / M[T]

STORE X ; Store final result in X

Explanation:

* The right part F(G+H)F(G+H)F(G+H) is calculated first and stored in T.
* Then the left part A[B+C(D+E)]A[B + C(D + E)]A[B+C(D+E)] is calculated using accumulator operations.
* Finally, the left result is divided by T to assign X.COA-Module-02-1.pptx

**4. Execution of X=A×B+C×CX = A \times B + C \times CX=A×B+C×C in One-Address, Two-Address, and Three-Address Processors**

* **One-Address Processor** (using accumulator):

text

LOAD A ; AC = A

MUL B ; AC = AC \* B

STORE T ; Store result in temporary T

LOAD C ; AC = C

MUL C ; AC = AC \* C

ADD T ; AC = AC + T

STORE X ; Store result in X

* **Two-Address Processor** (two operands per instruction):

text

MOV A, R1 ; R1 = A

MUL B, R1 ; R1 = R1 \* B

MOV C, R2 ; R2 = C

MUL C, R2 ; R2 = R2 \* C

ADD R2, R1 ; R1 = R1 + R2

MOV R1, X ; X = R1

* **Three-Address Processor** (three operands specified):

text

MUL A, B, R1 ; R1 = A \* B

MUL C, C, R2 ; R2 = C \* C

ADD R1, R2, X; X = R1 + R2

Explanation:

* As addresses increase, instructions become more flexible.
* One-address uses accumulator, two-address uses registers, and three-address provides full control over source and destination.COA-Module-02-1.pptx

**5. Stages Involved in the Basic Instruction Cycle (Detailed)**

The CPU executes instructions by going through these stages:

1. **Fetch**: The CPU reads the instruction from memory, with the address held in the Program Counter (PC).
2. **Decode**: The instruction is interpreted in the decoder to understand which operation to do.
3. **Operand Fetch**: If required, the CPU fetches the data operands from memory.
4. **Execute**: The CPU performs the arithmetic or logical operation using the ALU.
5. **Store Result**: The computed result is saved in a register or memory.
6. **Update PC**: The PC is updated to point to the next instruction.

This cycle repeats continuously until a halt or interrupt occurs.COA-Module-02-1.pptx

**6. Instruction Cycle State Diagram with Interrupts**

* The CPU normally cycles through Fetch → Decode → Execute states.
* When an **interrupt** occurs:
  + The CPU completes the current instruction.
  + Saves the current PC and status.
  + Transfers control to the interrupt service routine (ISR).
  + After ISR completes, restores PC and returns to the previous task.

This ensures the CPU responds to external events without losing the current program state.[mmrcse.blogspot](http://mmrcse.blogspot.com/2018/03/describe-instruction-cycle-state.html)

**7. Flynn’s Classifications**

Flynn's taxonomy classifies computer architectures based on instruction and data streams:

* **SISD (Single Instruction Single Data)**: Traditional serial computers; one instruction processes one data element at a time.
* **SIMD (Single Instruction Multiple Data)**: One control unit issues one instruction to multiple processing units working on different data elements simultaneously. Example: vector processors.
* **MISD (Multiple Instruction Single Data)**: Multiple instructions operate on the same data; uncommon in practice.
* **MIMD (Multiple Instruction Multiple Data)**: Multiple processors execute different instructions on different data streams independently. Used in modern multiprocessor systems.COA-Module-02-1.pptx

**8. Concept of Data Parallelism and Its Relationship to Flynn's Classifications**

* **Data Parallelism** involves applying the same operation across multiple data elements simultaneously.
* It fits well within the **SIMD** classification, where one instruction controls several processing units to work on different data concurrently.
* Data parallelism increases computation speed by dividing data across multiple processing units executing the same instructions.COA-Module-02-1.pptx

**9. Six-Stage Pipelining**

Pipelining lets the CPU process multiple instructions simultaneously by dividing instruction execution into stages:

1. **Instruction Fetch (IF)**
2. **Instruction Decode/Register Fetch (ID)**
3. **Operand Fetch (OF)**
4. **Execute (EX)**
5. **Memory Access (MA)**
6. **Write Back (WB)**

Each stage is handled by different hardware parts concurrently, like an assembly line, increasing throughput by overlapping instruction executions.COA-Module-02-1.pptx

**10. Control Unit: Softwired (Microprogrammed Control Unit)**

* The control signals for operations are stored as microinstructions in a special control memory (ROM).
* Each machine instruction is implemented as a sequence of microinstructions.
* The control unit fetches microinstructions from control memory to generate required control signals.
* Offers easy modification by changing microinstructions without altering hardware.
* Slower compared to hardwired due to microinstruction fetch overhead.COA-Module-02-1.pptx

**11. Control Unit: Hardwired Control Unit**

* Uses fixed combinational logic circuits to generate control signals directly based on the instruction opcode and timing pulses.
* Very fast because signals are generated by hardware paths.
* Complex to design for large instruction sets.
* Difficult to modify or add new instructions after design.COA-Module-02-1.pptx

**12. Differences Between Softwired (Microprogrammed) and Hardwired Control Units**

| **Feature** | **Hardwired Control Unit** | **Microprogrammed (Softwired) Control Unit** |
| --- | --- | --- |
| Implementation | Fixed combinational logic | Microinstructions stored in control memory |
| Speed | Faster | Slower due to microinstruction fetch time |
| Design Complexity | Complex and inflexible | Easier to design and modify |
| Flexibility | Limited | High flexibility to add/modify instructions |
| Examples | RISC processors | CISC processors, emulators |
| Modification | Difficult and costly | Simple as only microprogram changes are needed |

Here is a detailed and clear explanation based on the attached Module 3 notes:

**1. Digital Circuits: Basic Logic Gates**

* **NOT Gate (Inverter)**
  + Input 0 → Output 1, Input 1 → Output 0
  + It simply inverts the input signal.
* **AND Gate**
  + Output is 1 if **all** inputs are 1; otherwise, 0.
  + Example: Inputs (1,1) → Output 1; Inputs (1,0) → Output 0.
* **OR Gate**
  + Output is 1 if **at least one** input is 1; otherwise, 0.
  + Example: Inputs (0,1) → Output 1; Inputs (0,0) → Output 0.
* **NAND Gate**
  + Output is **inverted AND** output.
  + Output is 0 only if all inputs are 1; otherwise, 1.
* **NOR Gate**
  + Output is **inverted OR** output.
  + Output is 1 only if all inputs are 0; otherwise, 0.
* **EX-OR (Exclusive OR) Gate**
  + Output is 1 if inputs are **different**; otherwise, 0.
  + Example: Inputs (0,1) → Output 1; Inputs (1,1) → Output 0.
* **EX-NOR (Exclusive NOR) Gate**
  + Output is 1 if inputs are **same**; otherwise, 0.
  + It is the inverse of EX-OR.

**2. Flowchart for Booth's Multiplication Algorithm**

Booth's Algorithm efficiently multiplies signed binary numbers using these main steps:

* **Initialization**:  
  Set Accumulator = 0, Multiplier (Q), Multiplicand (M), Previous bit Q-1 = 0, and Counter = number of bits in multiplier.
* **Evaluate Pair (Q0, Q-1)**:
  + If 01 → Add M to accumulator
  + If 10 → Subtract M from accumulator
  + If 00 or 11 → No operation
* **Arithmetic Right Shift**:  
  Shift Acc, Q, Q-1 right by one bit, preserving the sign bit of Acc.
* **Decrement Counter**.
* **Repeat**:  
  Repeat evaluating and shifting until the counter reaches 0.
* **Result**:  
  The final product is stored in the combined Acc and Q registers.

This algorithm reduces the number of addition and subtraction steps compared to simple multiplication, making it faster especially for signed numbers.

**3. Number System Conversions and Examples**

* **Binary Number System**: Base 2 with digits {0,1}  
  Example: Decimal 19 = Binary 10011
* **Decimal Number System**: Base 10 with digits {0-9}
* **Octal Number System**: Base 8 with digits {0-7}  
  Often used for compact representation of binary numbers.
* **Hexadecimal Number System**: Base 16 with digits {0-9, A-F}  
  Also a compact form often used in programming.

**Conversions:**

* **Decimal to Binary, Octal, Hexadecimal**:  
  Use repeated division by the base and record remainders.
* **Binary to Decimal**:  
  Sum powers of 2 multiplied by each bit's value.
* **Binary to Octal/Hex**:  
  Group bits into sets of 3 (octal) or 4 (hex) and convert each group.