
# Computer Organization & Architecture

*Cybersecurity Department*

*Course Code:* ***CBS219***

***Lecture 4: CPU Organization: Control, ALU, & Registers***

Halal Abdulrahman Ahmed

# Lecture Outline

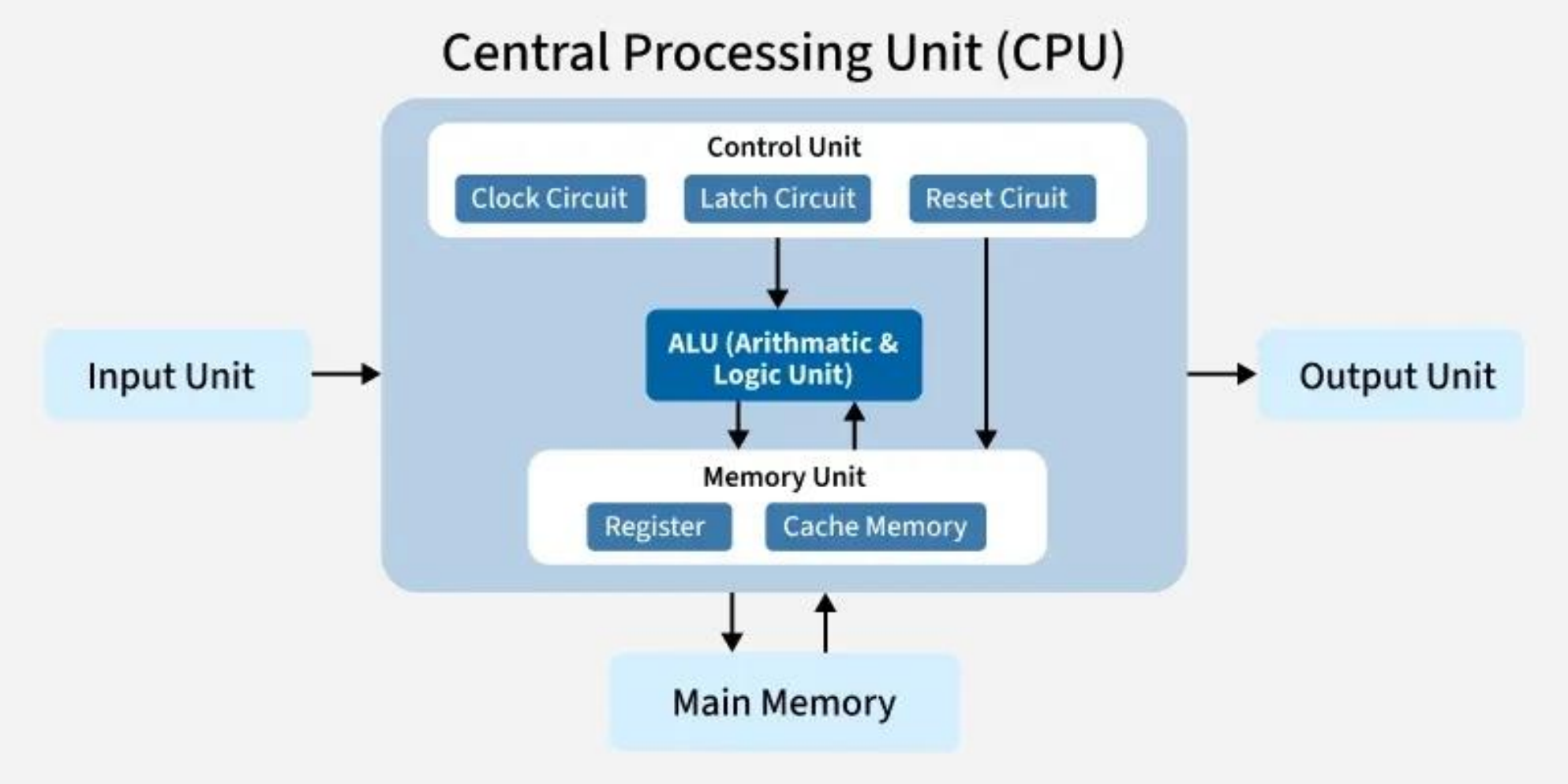
- Overview of CPU Components
- Control Unit (CU) & Instruction Control
- Control Signals & Execution Steps
- Status Flags & CPU Decisions
- System Clock & Synchronization
- Arithmetic Logic Unit (ALU)
- Arithmetic & Logical Operations
- Shift & Rotate Instructions
- Registers & Their Types

# Lecture Outcomes

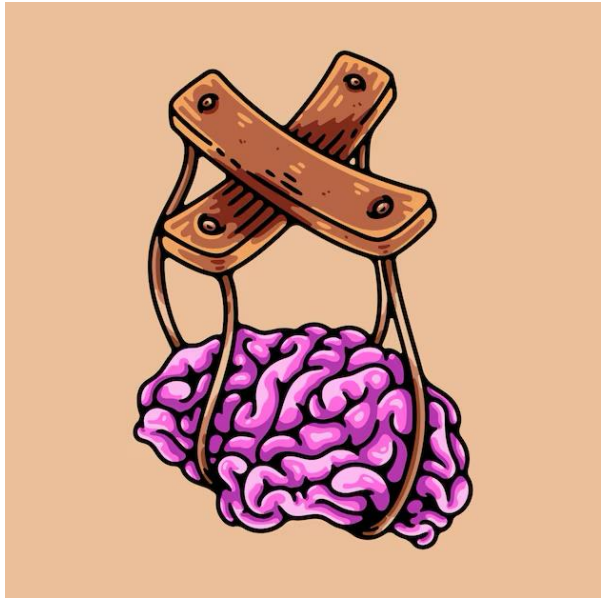
**By the end of this lecture, students will be able to:**

- Identify main internal units of the CPU (CU, ALU, Registers, Clock)
- Explain the role of the Control Unit in instruction execution
- Describe control signals and their purpose
- Understand and interpret CPU flags (Zero, Carry, Sign, Overflow)
- Explain the function of the system clock in CPU timing
- Understand ALU operations: arithmetic, logical, shifts, rotates
- Recognize different types of registers and their roles

# What is Inside the CPU?



# Control Unit



# Main Components of CPU

- **Control Unit:** The Control Unit controls and coordinates all activities inside the CPU. It sends control signals to other components and uses the system clock to manage timing, but it does not generate the clock. It makes sure each operation happens in the correct order. For example, it controls the movement of data from cache or registers to the ALU when needed.
- **Arithmetic and Logic Unit (ALU)**: The ALU handles arithmetic tasks (like addition, subtraction, multiplication, division) and logical tasks (like AND, OR, comparisons). It uses addition for all calculations.
- **Memory Unit**: The memory unit stores data and instructions. Older CPUs used registers, but modern ones also have fast cache memory. The CPU fetches data from RAM, ROM, or hard disks and stores it in registers or cache during tasks.

# Introduction to the Control Unit

- The Control Unit (CU) is the **part of the CPU that controls everything happening inside the computer** during instruction execution. It does **not perform calculations** like the ALU.
- Instead, it **gives orders** to other CPU parts on **what to do, when to do it, and how to do it**. It reads instructions from memory **one by one** and **makes sure they are executed correctly**.
- It acts like a **traffic controller** inside the CPU making sure data goes to the right place at the right time.

# Why is the Control Unit Needed?

- Without the Control Unit, the CPU would **not know what to do with instructions**.
- The CU ensures:
  - Every instruction is **executed step by step**.
  - The CPU **fetches the correct instruction from memory**.
  - The ALU knows **which operation to perform**.
  - The registers know **where to move data**.
  - Memory receives **read/write commands**.
- It also **synchronizes operations** using clock pulses (timing).

# Main Jobs of the Control Unit

- The Control Unit has **five main responsibilities**:
- **Fetch instructions** from memory.
- **Decode instructions** to understand what they do.
- **Send control signals** to execute instructions.
- **Manage program order** using the Program Counter (PC).
- **Control data movement** between registers, ALU, and memory.

# What are Control Signals?

- Control signals are **electrical commands** sent by the CU.
- They **tell other parts of the CPU what to do**.

Examples:

- READ → Get data from memory.
- WRITE → Store data in memory.
- LOAD R1 → Load data into register R1.
- ALU = ADD → Tell ALU to perform addition.
- PC = PC + 1 → Move to next instruction.

**Without control signals**, hardware like ALU and registers will just **sit idle and do nothing**.

# How Control Signals Work (Simple Example)

Imagine this instruction: `ADD R1, R2`

- This means: **R1 = R1 + R2**

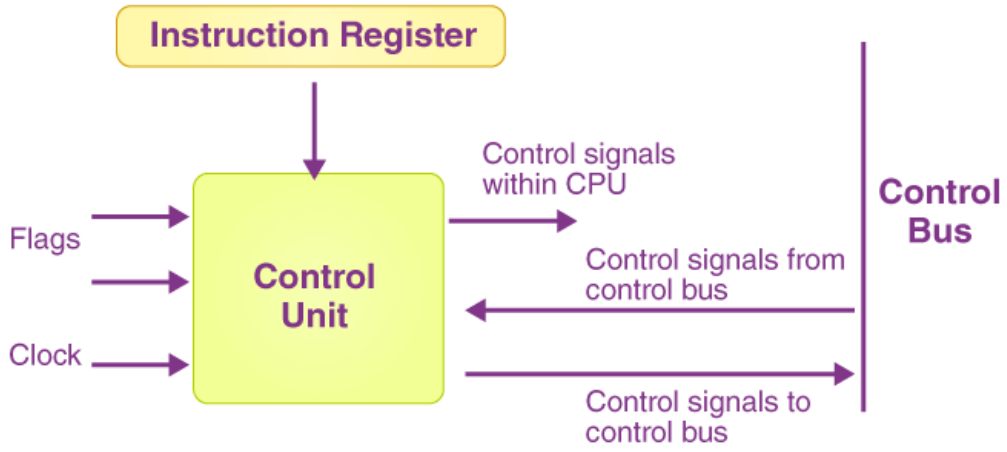
To do this, the Control Unit sends:

- Signal to **read values** from register R1 and R2.
- Signal to **send values to ALU**.
- Signal to **set ALU to ADD mode**.
- Signal to **store result back in R1**.
- Signal to **update flags** (Zero, Carry, Overflow).

So this **one instruction** needs **many internal steps** controlled by CU.

# How Does a CPU Control Unit work?

A control unit receives data from the user and translates it into control signals that are subsequently delivered to the central processor. The processor of the computer then instructs the associated hardware on what operations to do. Because CPU architecture differs from manufacturer to manufacturer, the functions performed by a control unit in a computer are dependent on the CPU type.




Block Diagram of the Control Unit

# Flag (Status Register)

- The **Flag Register**, also called the **Status Register** or **Condition Code Register**, is a **special CPU register** that stores **status information** about the result of the most recent ALU operation.
- Each **flag is 1 bit** (0 or 1) and is automatically **set (1) or cleared (0)** by the ALU.
- The **Control Unit reads these flags** to decide what to do next during program execution, especially for **conditional instructions** such as jumps and loops.
- Flags help the CPU **make decisions** based on results.
- Flags allow the Control Unit to control program flow using instructions like JZ (Jump if Zero), JC (Jump if Carry), JNZ (Jump if Not Zero), etc.

# Flag (Status Register)

| Flag | Meaning | When it is set (1) |
|---|---|---|
| **Z – Zero Flag** | Result is zero | If operation result = 0 |
| **C – Carry Flag** | Carry or borrow occurred | In unsigned arithmetic overflow |
| **S – Sign Flag** | Result is negative | If MSB (most significant bit) = 1 |
| **O – Overflow Flag** | Arithmetic overflow | In signed arithmetic overflow |

**Example:**

If the CPU executes 5 − 5, the result is 0, so **Zero Flag (Z) = 1**.

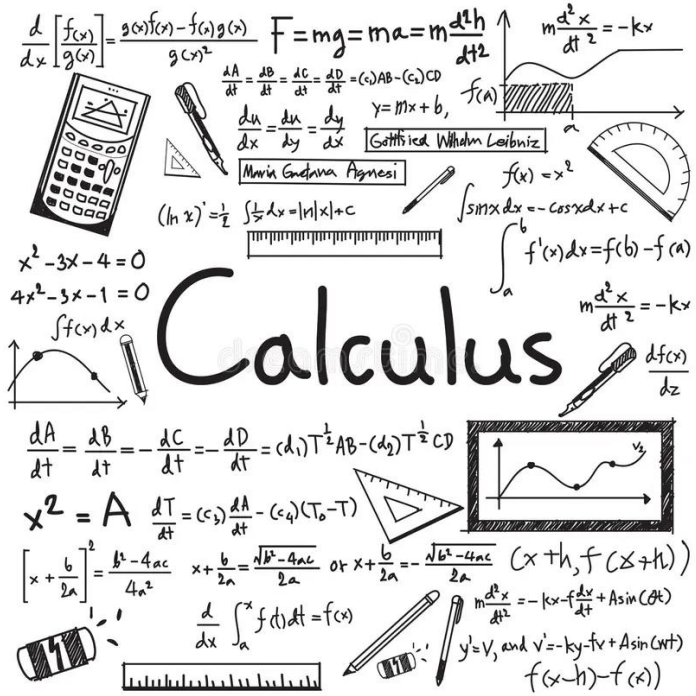
If CPU executes 200 + 100 in 8-bit, result exceeds 255 → **Carry Flag = 1**.

If result = –3, binary begins with 1 → **Sign Flag = 1**.

# Clock

- The clock is like the **heartbeat** of the CPU.
- It sends regular electronic pulses that control the timing of everything inside the CPU.
- Each pulse is called a **clock cycle**.
- Every step the CPU does (fetch, decode, execute) happens on a clock cycle.
- The Control Unit uses the clock to do things step by step in the correct order.
- Without the clock, the CPU would not work because nothing would happen in the right time.

# Arithmetic Logic Unit (ALU)



# Introduction to Arithmetic Logic Unit (ALU)

- The ALU is a digital circuit within the CPU that performs all **arithmetic** **and** **logical** **operations**. It takes **input** from **registers**, **processes the operation**, and **sends the result back** to a destination **register or memory**.
- The ALU takes the input **operands** and an **instruction** and outputs the result.
- Performs operations like **addition, subtraction, AND, OR, NOT,** etc.
- **Controlled by** the **control unit** based on the instruction type.

# Introduction to Arithmetic Logic Unit (cont.)

- In some processors, the ALU is divided into two units, an arithmetic unit **(AU)** and a logic unit **(LU).** Some processors contain more than one AU for example, one for **fixed-point operations** and another for **floating-point operations**.

- The ALU is a **vital part of modern** CPUs. Most CPUs contain many subcomponents for various functions, including accumulators, registers, a memory manager, an ALU, a floating-point unit (FPU) and cache memory. New processors may also have a graphics processing unit (GPU) and a neural processing unit (NPU).

# What is the purpose of an arithmetic logic unit?

- The purpose of an ALU is to speed up a CPU's overall processing by performing math and logic functions.
- By splitting out these functions, the different portions of the CPU can be more specialized and perform different operations simultaneously.

# What is the purpose of an arithmetic logic unit? (cont.)

- In early microprocessors, the main CPU could only perform basic operations, and more complex processes, like math, required many steps and took a long time to perform. A separate chip, often called a ***math coprocessor***, could be added to offload these slow functions so the CPU could perform other work.
- Modern CPU cores use a **pipelining** approach to work on multiple things at the same time. With pipelining, for example, the memory manager can load items into registers while the ALU is performing an add operation.

# How does an arithmetic logic unit work?

- Typically, the ALU has direct access to the processor controller, main memory (RAM in a PC) and the input/output (I/O) of the CPU. I/O flows along an **electronic path** called a **bus**.

- The input consists of an **instruction word**, sometimes called a **machine instruction word**, that contains an **operation code (opcode),** one or more operands, and sometimes a **format code**.

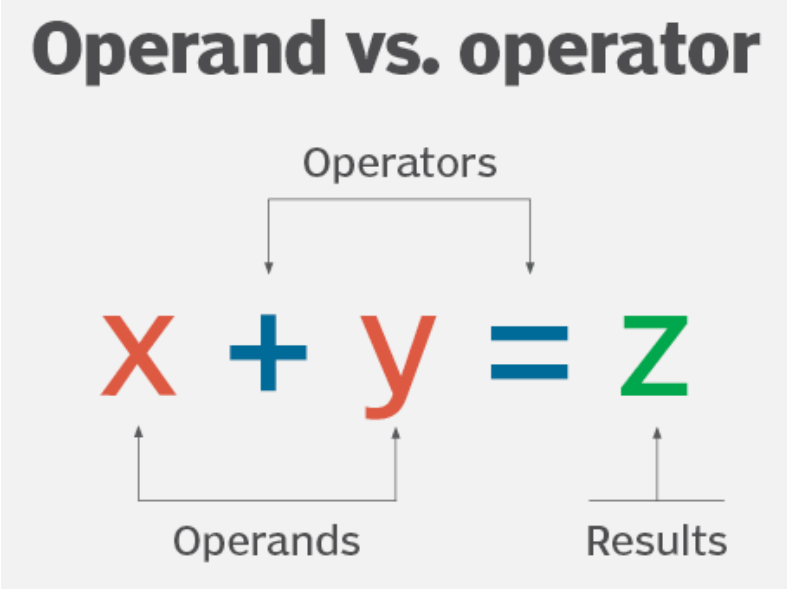
- The opcode tells the ALU what **operation** to perform, and the operands are used in the operation.

When the CPU executes an instruction:

•It takes **two inputs (operands)**, like numbers stored in registers.

•It performs an **operation** on them, such as **addition** or **comparison**.

•The **instruction format** tells the CPU **how to treat the data,** as normal integers (**fixed-point**) or as decimal numbers (**floating-point**).

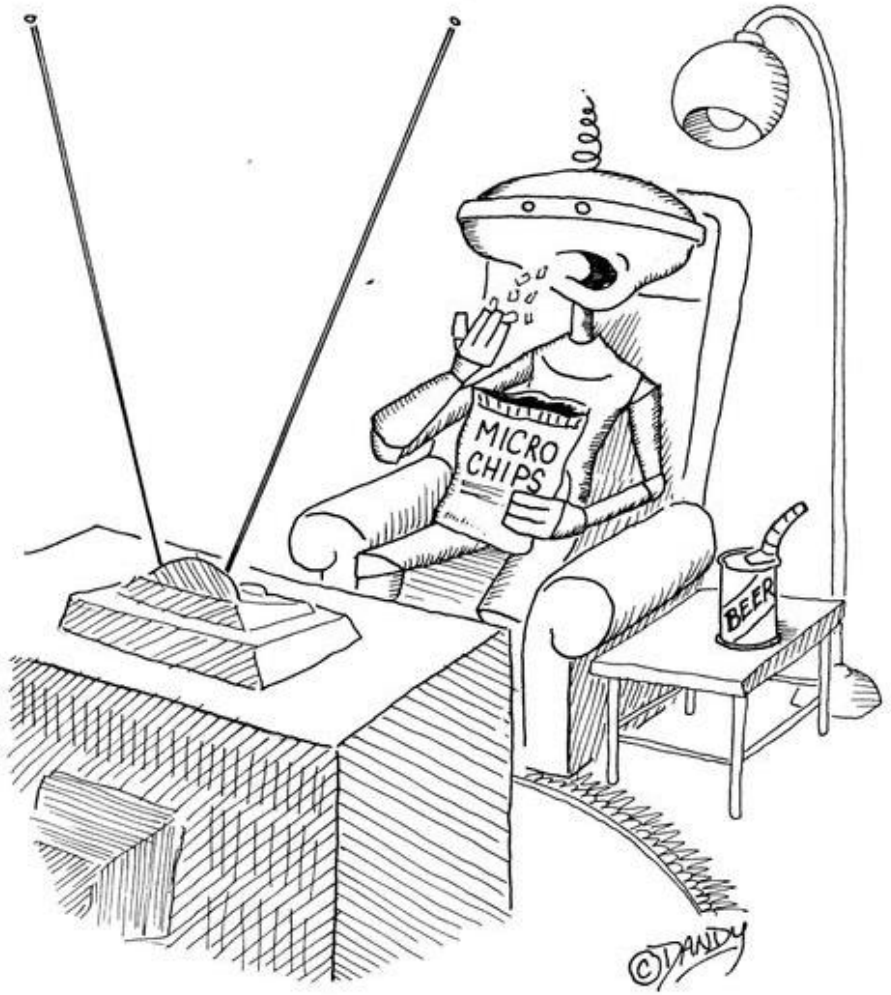



Steps:

- Take operand 1 from **X**
- Take operand 2 from **Y**
- Perform addition → **X + Y = Z**
- Store the result back in **Z**
- Update flags (Zero, Carry, Overflow, etc.)
- If everything worked, no error is stored.

# Functions and Operations of ALU

1. **Arithmetic Operations**
2. **Logical Operations**
3. **Shift Operations**



# Arithmetic Operations

# Arithmetic Operations

- ALU performs arithmetic operations to carry out basic mathematical calculations on binary numbers. Operations include addition, subtraction, increment, and decrement.

- **Multiplication and division** are arithmetic operations, but **multiplication and division are not always performed by the basic ALU**. In modern CPUs, they are usually done by a **separate arithmetic unit** (Multiplier/Divider Unit) that works **alongside** the ALU. It is a **separate hardware unit** called the **Multiplier/Divider Unit** (part of an extended ALU or Execution Unit), especially in modern CPUs.

# Arithmetic Operations (cont.)

- Used during instruction execution for tasks like ADD, SUB, INC, and DEC.
- **Addition** → adds two numbers
- **Subtraction** → subtracts one number from another
- **Increment** → increases a number by 1
- **Decrement** → decreases a number by 1

# Logical Operations

# Logical Operations

- Logical operations in the ALU work with data **at the bit level (changing or checking individual bits)** using **logic gates** such as AND, OR, XOR, and NOT. These operations are actually built using **logic gates inside the ALU hardware**.
- Logical operations are not mathematical; instead, they are used to **control or modify individual bits** inside a binary number.

# Logical Operations

- They are important for tasks like checking certain bits, enabling or disabling features, setting flags, or controlling hardware.
- These operations are done on binary data and are essential in low-level programming, computer hardware control, security algorithms, and instruction execution.

# Logical Operations

- **AND** → Output is 1 **only if both bits are 1**. Used for **masking** (keeping selected bits).
- **OR** → Output is 1 **if any bit is 1**. Used to **set specific bits**.
- **XOR (Exclusive OR)** → Output is 1 **if bits are different**. Used for **toggling bits** or fast comparisons.
- **NOT** → **Flips bits** (1 becomes 0, 0 becomes 1). Used to **invert** data.

# AND Operation

- **Rule:** Output is **1 only if both bits are 1**

  **Use:** Used for **masking** → keeping only selected bits and turning others off

| Bit A | Bit B | A AND B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | **1** |

# AND Operation

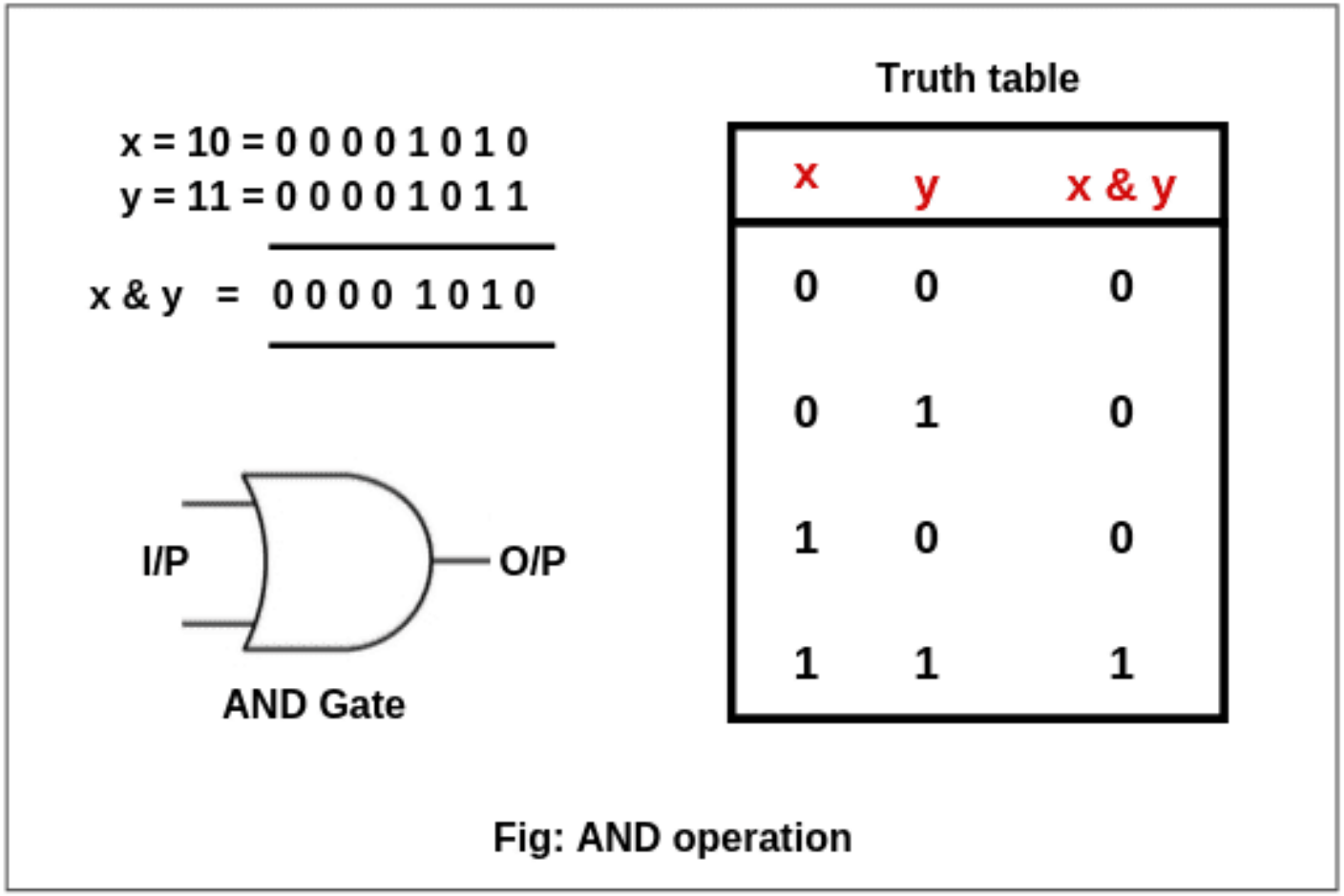



Fig: AND operation

# OR Operation

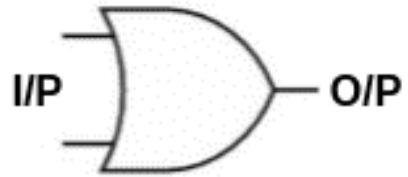
- **Rule:** Output is **1 if any bit is 1**
- **Use:** Used to **set specific bits** (turn bits ON)

| Bit A | Bit B | A OR B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

# OR Operation Example

x = 20 = 0 0 0 0 0 1 0 1
y = 10 = 0 0 0 0 1 0 1 0

x | y = 0 0 0 0 1 1 1 1




OR Gate

Truth table

| x | y | x \| y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Fig: OR operation

# XOR (Exclusive OR)

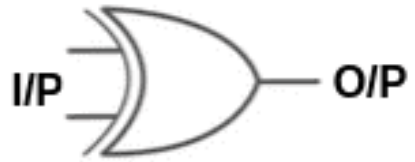
- **Rule:** Output is **1 only if bits are different**
- **Use: Toggle bits** (flip them) or fast comparisons

| Bit A | Bit B | A XOR B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

# XOR (Exclusive OR)

x = 20 = 0 0 0 0 0 1 0 1
y = 10 = 0 0 0 0 1 0 1 0

x ^ y = 0 0 0 0 1 1 1 1




XOR Gate

Truth table

| x | y | x ^ y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Fig: XOR operation

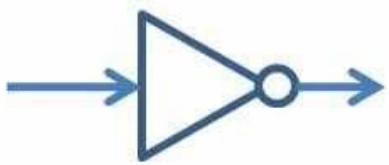
# NOT Operation

- **Rule: Flips bits** → 1 becomes 0, 0 becomes 1
- **Use: Invert** data

| Bit A | NOT A |
|---|---|
| 0 | 1 |
| 1 | 0 |

## NOT

| x | F |
|---|---|
| 0 | 1 |
| 1 | 0 |

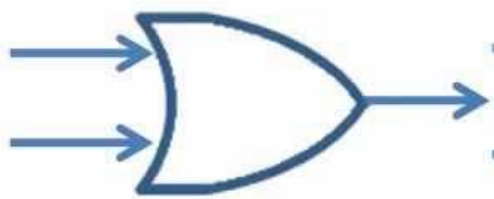


## AND

| x | y | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

## OR

| x | y | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |



## XOR

| x | y | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

# Uses of each logical gate:

- **AND gate: Used for masking**

Keeps only selected bits and hides the rest.

- **OR gate: Used for setting bits**

Turns specific bits ON (enables features or options).

- **XOR gate: Used for toggling bits**

Flips specific bits (used in switching and encryption).

- **NOT gate: Used for bit inversion**

Reverses all bits (used for creating negative or opposite values).

# Shift Operations

# Shift Operations

- Shift operations are basic bit manipulation operations performed inside the Arithmetic Logic Unit (ALU). They move bits to the left or right and are used in arithmetic, logic manipulation, and data processing.

Types include:

1. **Logical Shift**
2. **Arithmetic Shift**
3. **Rotations**

# What are Shift Operations?

- Shift operations move the bits of a binary number **left or right**.
- Each shift moves bits by a **specified number of positions**.
- Bits that are shifted out are **lost**, and empty bit positions are filled based on the **type of shift**.

# Shift Operations

Shift operations are widely used in computer systems because they are:

- **Fast** (performed directly in hardware by the ALU)
- **Efficient** for:
  - Multiplication and division by powers of **2.**
  - Bit extraction from binary data (Example: If we want to get only the last 3 bits of a number, we can shift and mask it easily)
  - Encoding and decoding operations, such as converting characters or signals into binary form and back again. It's used in communication systems and data transmission.
  - Cryptography and data compression (Example: XOR + bit shifting is used in simple encryption techniques)

# Types of Logical Shift

1. Logical Shift Left (LSL)
2. Logical Shift Right (LSR)

# Logical Shift Left (LSL)

- Moves all bits to the **left** by n positions.
- **Zeros** are inserted from the **right**.
- Used for **unsigned multiplication** by 2.

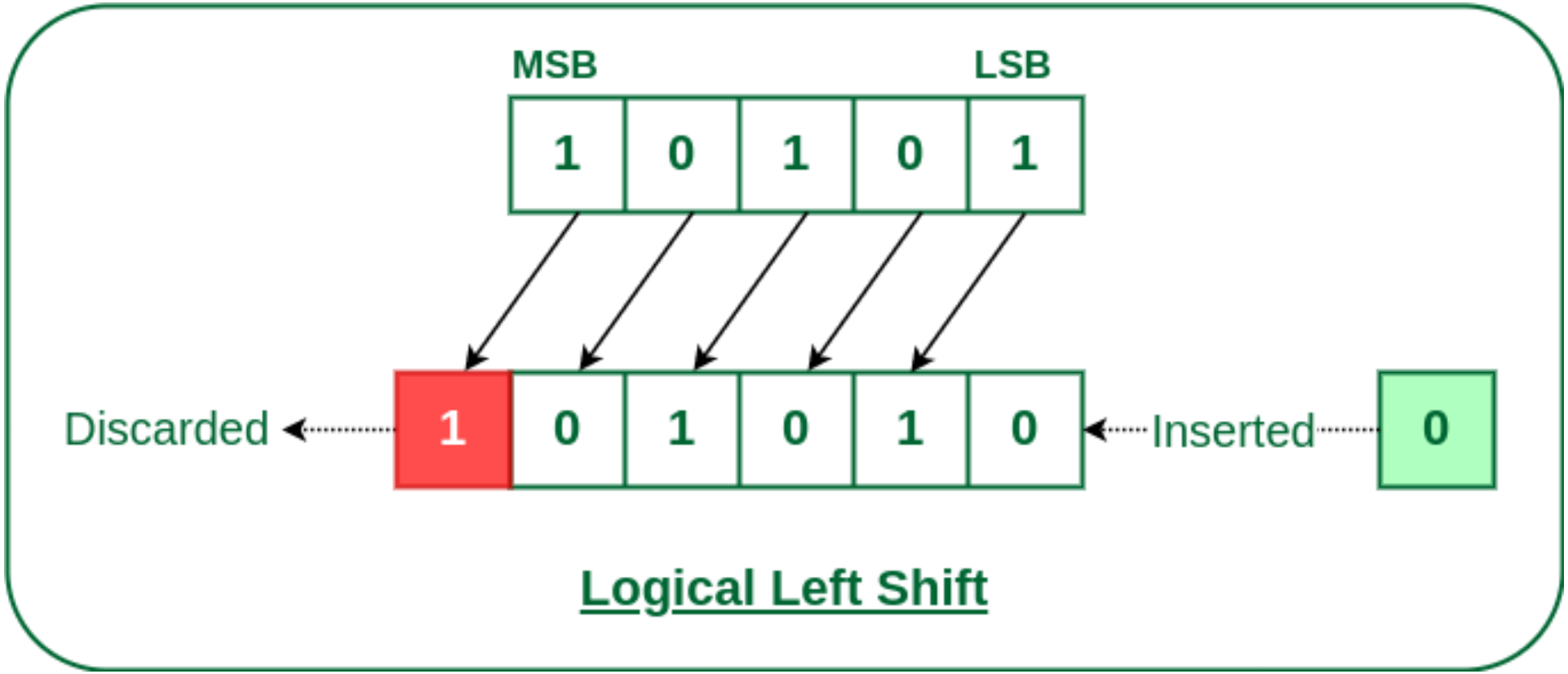
Example:

Binary: 0011 0101 (53)

LSL by 1 → 0110 1010 (106)

- Each bit shifted left.
- A 0 is inserted at the right.
- Number doubled.

# Logical Shift Left (LSL)



# Examples of Logical Shift Left (LSL)

Let's take an 8-bit unsigned binary number **01010011** (which is **83** in decimal). If we perform a logical shift left:

- **Before Shift:** 01010011 (83 in decimal)
- **After Logical Shift Left:** 10100110 (166 in decimal)

**Example:**

0010 1100 (44 in decimal)

Shift Left by 1 → 0101 1000 (88) → multiplied by 2

# Logical Shift Right (LSR)

- Moves all bits to the **right** by n positions.
- **Zeros** are inserted from the **left**.
- Used for **unsigned division** by 2.

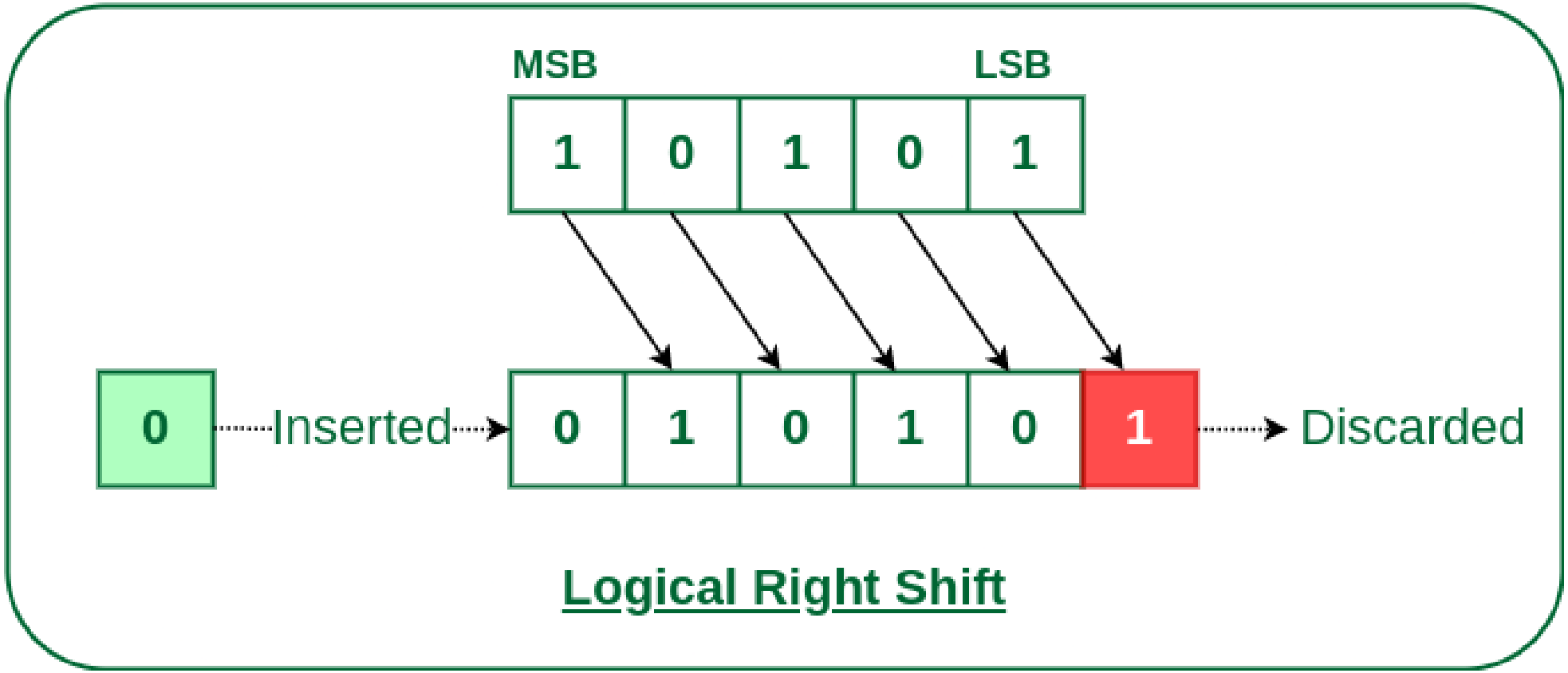
Example:

Binary: 0011 0101 (53)

LSR by 1 → 0001 1010 (26)

- Right shift removes last bit.
- New 0 is added to left.
- Number divided by 2.

# Logical Shift Right (LSR)



# Example of Logical Shift Right (LSR)

Let's take the same 8-bit binary number 01010011 (83 in decimal). If we perform a logical shift right:

- **Before Shift:** 01010011 (83 in decimal)
- **After Logical Shift Right:** **0**0101001 (41 in decimal)

# Arithmetic Shift

- **Arithmetic Shift** is used for **signed binary numbers (two's complement)**. It shifts bits left or right **while preserving the sign of the number**.
- Preserves the **sign bit (MSB)** during shifting.
- Used for **fast multiplication or division by 2**.
- Two types:
  - **Arithmetic Shift Left (ASL)** → multiply by 2
  - **Arithmetic Shift Right (ASR)** → divide by 2 and keep sign
- In most CPUs, **ASL behaves the same as LSL** (Logical Shift Left), so it may not appear as a separate instruction.

# Arithmetic Shift Left (ASL)

- Shifts all bits **one position to the left**.
- **Sign bit may change** → can cause **overflow**.
- A **0** is inserted from the **right**.
- Used for **multiplying a signed number by 2**.

# Examples of Arithmetic Shift Left (ASL)

+22 in binary (8-bit signed): 0001 0110

ASL by 1 → 0010 1100 = +44

-10 in binary (8-bit signed): 1111 0110

ASL by 1 → 1110 1100 = -20

- Shift left → multiply by 2
- For negative values, sign bit remains 1 until overflow happens
- Some processors simply use **LSL instead of ASL**

# Arithmetic Shift Right (ASR)

- Shifts all bits **one position to the right**.
- **Preserves the sign bit** (MSB stays the same).
- Used for **signed division by 2**.
- Prevents incorrect sign changes.

# Arithmetic Shift Right (ASR)

- **Example 1 – Positive number:**

ASR by 1 → 0001 0110 (+22)

0000 1011 (+11)

- **Example 2 – Negative number:**

ASR by 1 → 1111 0110 (-10)

1111 1011 (-5)

Before ASR:
1 1 1 1 0 1 1 0
↑sign bit = 1

Shift Right:
1 1 1 1 1 0 1 1
↑new bit added = same as sign bit (1)

# Rule for Arithmetic Shift Right (ASR)

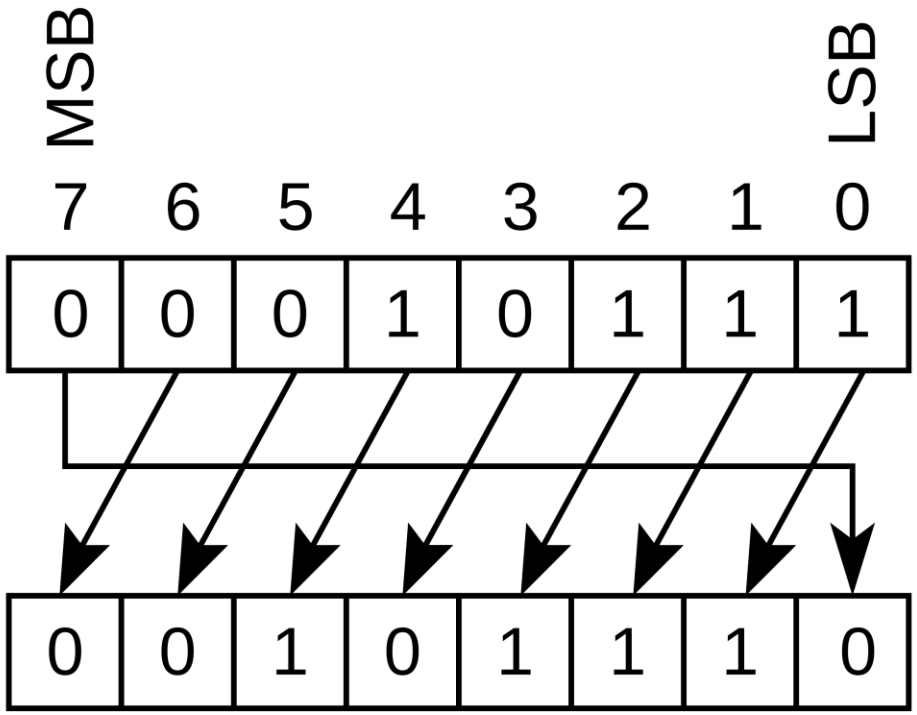
- When shifting right:
- If the **most significant bit (MSB) = 0** → we insert **0** from the left (number stays positive)
- If the **most significant bit (MSB) = 1** → we insert **1** from the left (number stays negative)

# Rotate Operations

- Rotate (circular shift) is a bit manipulation operation in the ALU where bits are shifted left or right, and **the bit that falls off from one end is reinserted at the other end**.

# Rotate Left (ROL)

- Bits are shifted left.
- The **leftmost bit** wraps around to the **rightmost** position.
- **No bit is lost**.



# Example of Rotate Left (ROL)

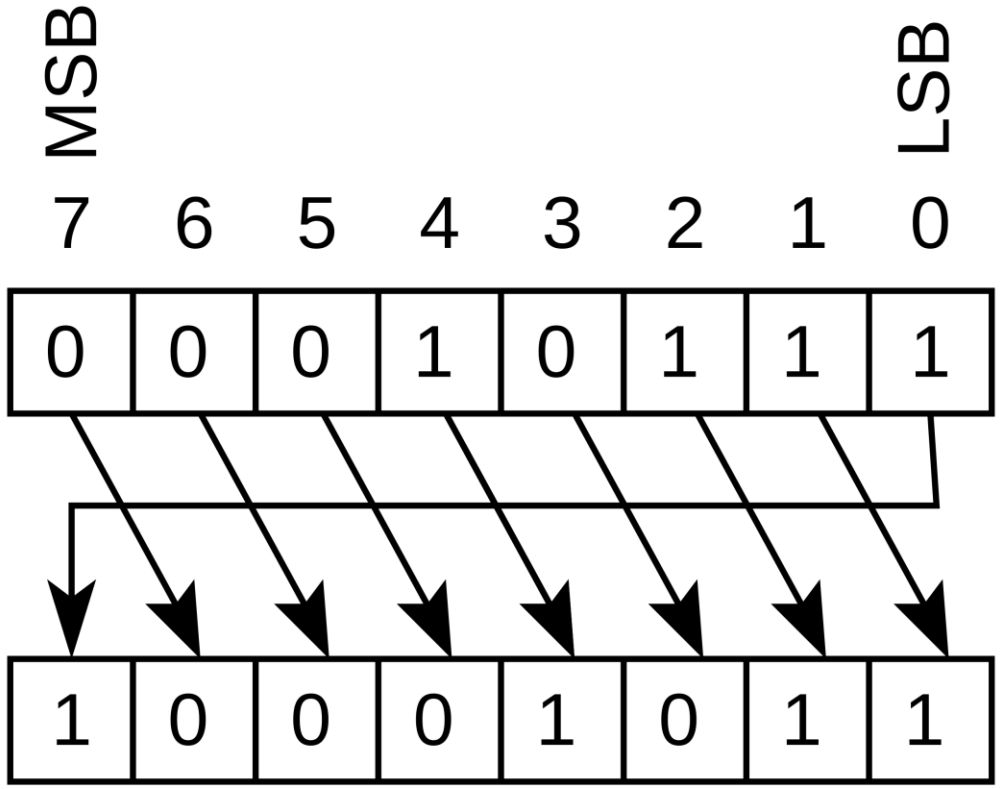
- Before: 1011 0010
- ROL by 1:
- Step 1: shift left → 0110 010?
- Step 2: reinsert MSB → 0110 0101
- Result: 0110 0101

# Example of Rotate Left (ROL)

- Binary: 1011 0010
- ROL by 1 → 0110 0101
- Leftmost bit (1) moves to the end.
- Circular shift.

# Rotate Right (ROR)

- Bits are shifted right.
- The **rightmost bit** wraps around to the **leftmost** position.



# Example of Rotate Right (ROR)

Before: 0110 1001

- ROR by 1:
- Step 1: shift right → ?011 0100
- Step 2: reinsert LSB → 1011 0100
- Result: 1011 0100

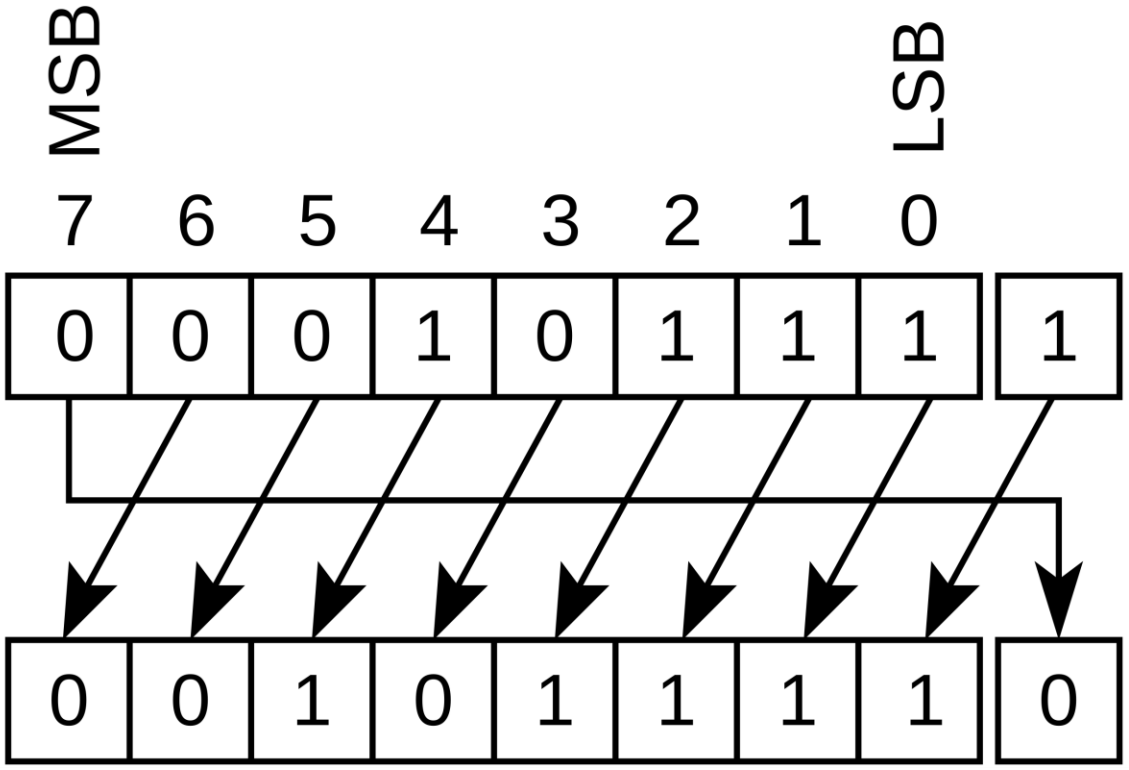
- Binary: 1011 0010
- ROR by 1 → 0101 1001

- Last bit moves to the beginning.
- No data is lost.

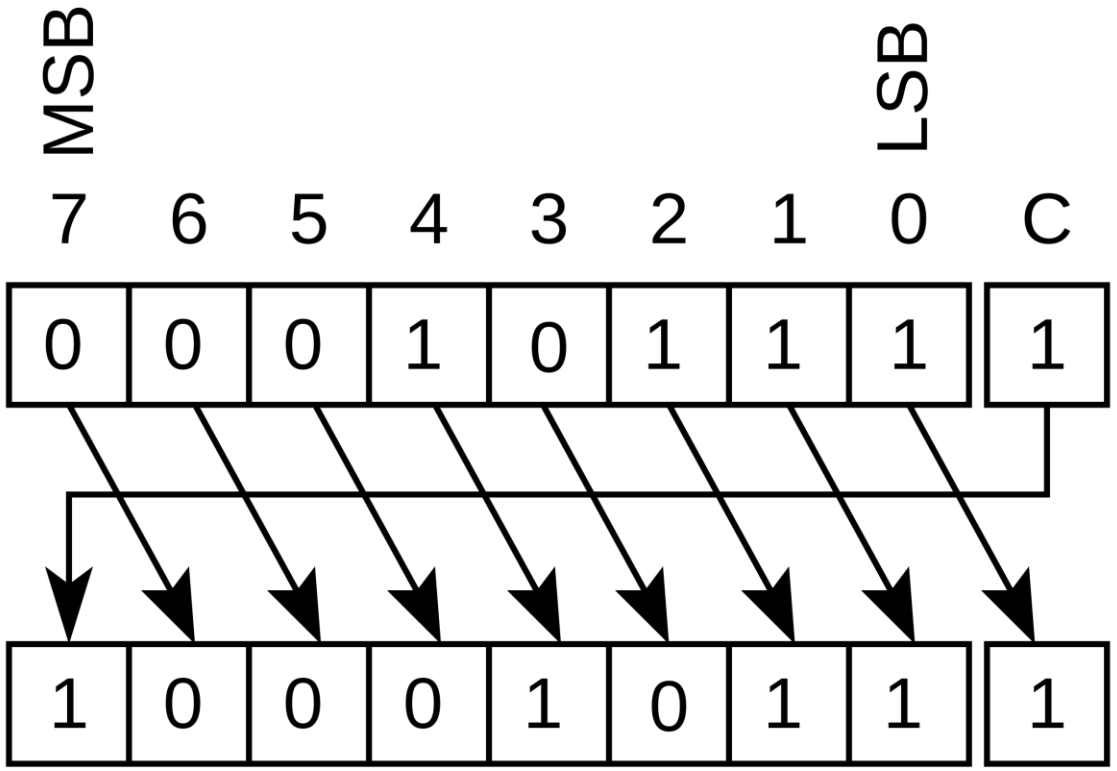
# Rotate Through Carry (RCL / RCR)

- Rotate Through Carry shifts bits **including the Carry Flag (C)** in the rotation. Used in low-level assembly.
- Uses **9 bits** in rotation (8 bits + Carry flag).
- Allows **multi-word shifts** and password hashing in low-level code. When shifting more than 8 bits (like 16-bit or 32-bit numbers), rotate through carry helps connect multiple registers together.
- It is used in password hashing / encryption
- Bit rotation is used in many security algorithms and cryptography.
- Rotating bits makes data difficult to guess or reverse, which increases security.

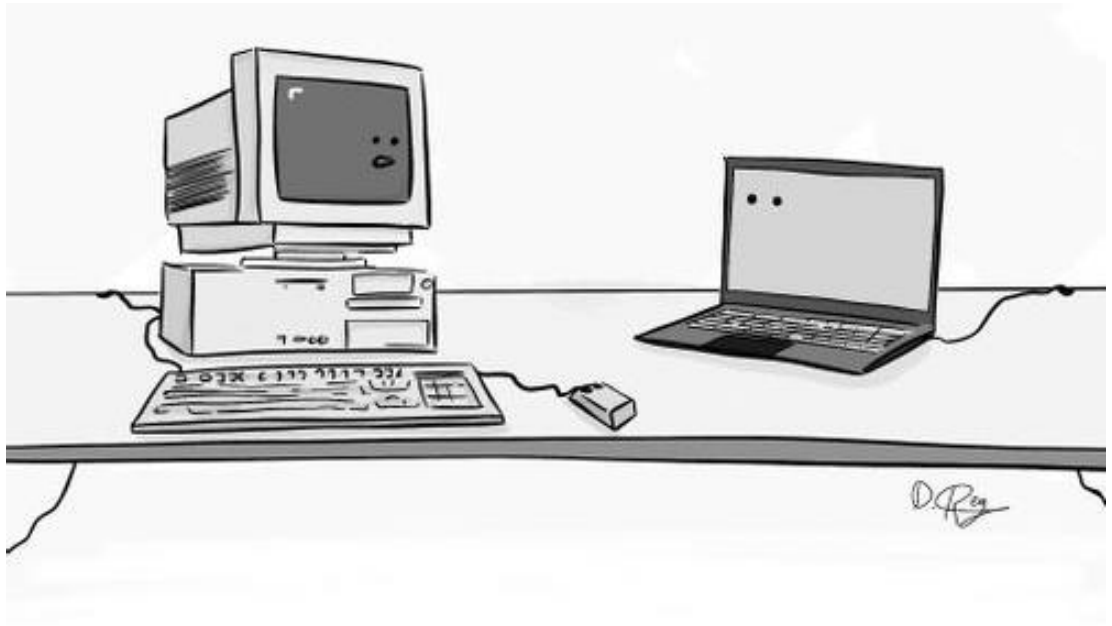
# Rotate Left Through Carry (RCR)



# Rotate Right Through Carry (RCL)



# Registers



*"Hey, can I borrow some cache?"*

# What is Register in Digital Electronics ?

- A register is a small and temporary storage unit inside a computer's (CPU). It plays a vital role in holding the data required by the CPU for immediate processing and is made up of flip-flops. It usually holds a limited amount of data ranging from 8 to 64 bits, depending on the processor architecture.
- Registers act as intermediate storage for data during arithmetic logic and other processing operations.

# What is Register?

- A register is a tiny, fast storage memory within the central processing unit (CPU) or the arithmetic logic unit (ALU) of a computer. Registers are utilized for a variety of functions in handling and controlling instructions and data and play an important role in the operation of a computer's CPU.
- In simple words, we can say that Register is very small, tiny storage unit which is temporary in nature but fast storage memory of a computer.

# Types of registers:

- Accumulator Register
- Program Counter (PC) Register
- General-Purpose Registers
- Instruction Register (IR)
- Memory Address Register (MAR)
- Memory Data Register (MDR)
- Stack Pointer (SP)
- Floating-Point Registers

# Types of Registers

- **Accumulator Register**

The accumulator acts as a central point for arithmetic and logical operations within the CPU. It fetches data from memory and stores intermediate results during calculations. Arithmetic operations such as addition, subtraction, multiplication, and division often take place in the accumulator. The final result may be stored in the accumulator or transferred to other registers or memory locations.

- **Program Counter (PC) Register**

The program counter is a special register that keeps track of the memory address of the next instruction to be fetched and executed. As the CPU executes each instruction in sequence the program counter is updated to indicate the next instruction's address in memory. This process continues until the program's execution is complete.

# Types of Registers (cont.)

- **General-Purpose Registers**

General-purpose registers are versatile because they can hold data and memory addresses. They are used for various calculations and data manipulation tasks during program execution. General-purpose registers are essential for performing arithmetic and logical operations on data stored in the CPU.

- **Instruction Register (IR)**

The instruction register holds the currently fetched instruction from memory. It allows the CPU to decode and execute the instruction based on its opcode and operands.

# Types of Registers (cont.)

- **Memory Address Register (MAR)**

The memory address register stores the memory address of data or instructions to be accessed or written in memory. It plays a crucial role in memory operations by indicating the location of the data or instruction the CPU needs to access.

- **Memory Data Register (MDR)**

The Memory Data Register holds the actual data fetched from or written to memory. When the CPU retrieves data from memory, it is temporarily stored in the MDR before being processed further.

# Types of Registers (cont.)

- **Stack Pointer (SP)**

The stack pointer is used in stack-based memory operations. It keeps track of the top of the stack. which is a region of memory used for temporary storage of data and return addresses during function calls.

- **Floating-Point Registers**

The Floating-point registers are specialized for handling floating-point numbers and performing floating-point arithmetic operations. These registers can store and manipulate floating-point numbers with higher precision.

# Types of Registers (cont.)

- **Index Register (IR)**

The Index Register is a kind of useful instrument in the computer that tracks specific locations of data. It is like a small helper that makes it easy for the computer to get to where memory is stored quickly and process it.

- **Memory Buffer Register (MBR)**

The Memory Buffer Register can be viewed as a temporary store in the computer where data takes time off when it is being transferred from one place to another by the CPU. This register can be seen as some kind of traffic police man, ensuring smooth movements of information between CPU and memory without any bumps.

# Types of Registers (cont.)

- **Data Register (DR)**

The Data Register is an example of a fast storage place in the computer where information rests while all these processes take place on the same machine. It's a location where transient amounts of data are kept within the system for temporary usage like pit stops within then performing calculations or tasks or others.

# Advantages of Registers

**Advantages of Registers**

- **Speed:** The Registers offer fast access times due to their proximity to the CPU, enhancing overall system performance.
- **Data Processing Efficiency:** They enable quick data manipulation, reducing the need to access slower main memory frequently.

# Disadvantages of Registers

- **Limited Capacity:** The Registers have a small size, restricting the amount of data they can hold at a time.
- **Cost:** The Registers are made from flip-flops and require more hardware, contributing to the overall cost of the processor.

# Further Learning Resources

- **COA Tutorials (Text)**: GeeksforGeeks – CPU Units Overview

- **Interactive Learning** (Video): CPU Architecture

  *Short and easy video showing how ALU, CU, and registers work together.*

# References


- Fox, C. (2022). Computer architecture: From the Stone Age to the quantum age. No Starch Press.
- Plantz, R. G. (2020). Introduction to computer organization: An under-the-hood look at hardware and ARM64 assembly (ARM ed.). No Starch Press.
- Stallings, W. (2022). Computer Organization and Architecture: Designing for Performance (11th ed.). Pearson.
- Boyd, C. (2020, January 1). How CPUs are designed and built: Fundamentals of computer architecture. TechSpot. https://www.techspot.com/article/1821-how-cpus-are-designed-and-built/


Any
Question