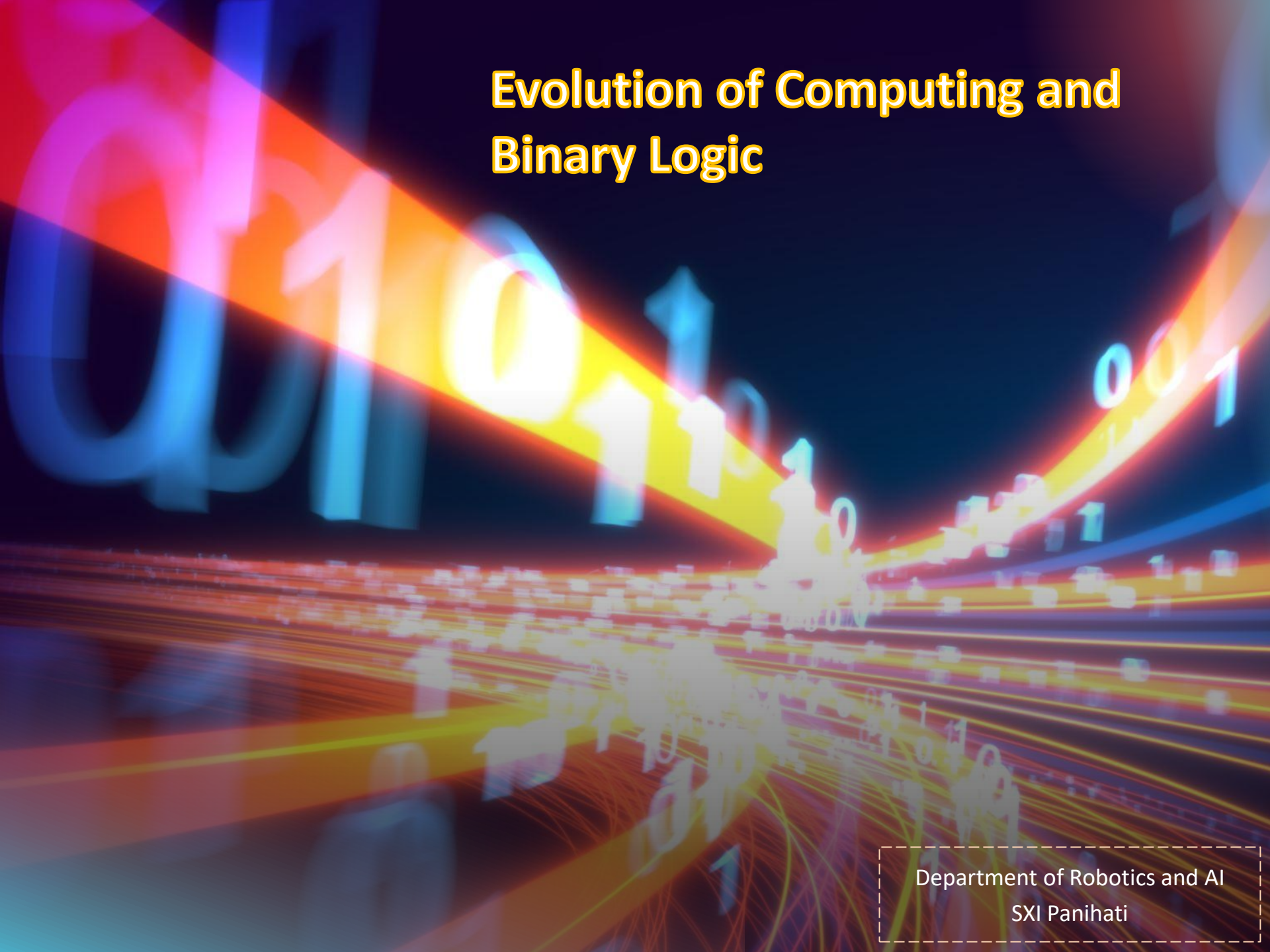
Evolution of Computing and Binary Logic



Department of Robotics and AI
SXI Panihati

Introduction and Phases of Evolution of Computing

- **Binary Logic and Early Computing**: Computers process data using binary (0s and 1s). Early machines relied on vacuum tubes and later transistors to execute binary instructions.
- **Low-Level Programming**: Early programming used machine language (binary instructions) and assembly language, which introduced symbolic codes for easier hardware control.
- **Operating Systems**: These enabled multitasking by managing hardware and allowing multiple programs to run simultaneously, improving efficiency and user interaction.
- **High-Level Programming Languages**: Languages like FORTRAN, COBOL, and Python simplified coding with human-readable syntax, making software development more accessible.
- **AI and Automation**: Modern computing enables AI-driven systems that learn, adapt, and automate tasks, powering innovations like voice assistants, self-driving cars, and robotics.

Importance of Binary Logic in Computing

- **Digital Circuits**: Logic gates, the building blocks of digital circuits, process binary signals (0 and 1) to perform operations. These gates form complex circuits used in computers, smartphones, and embedded systems, enabling everything from simple calculations to advanced processing.
- **Programming**: Binary logic plays a key role in programming, where decisions are made based on true/false conditions. Conditional statements and loops rely on binary decisions to control program flow, making software execution precise and efficient.
- **Decision-Making**: Binary logic is crucial in automated systems like traffic lights, industrial automation, and AI-driven applications. These systems operate based on high (1) or low (0) signals, ensuring reliable and consistent decision-making for real-world tasks.
- **Importance of Binary Logic**: The clear distinction between 0 and 1, true and false, or high and low signals makes binary logic highly reliable. This fundamental principle ensures accuracy in digital circuits, computing, and AI, forming the backbone of modern technology.

Introduction to Logic Gates

Logic gates are the building blocks of digital circuits that process binary inputs to produce specific outputs based on logical operations.

NOT Gate (Inverter)

A **NOT gate** is a fundamental digital logic gate that inverts its input.

If the input is **1** (High/True), the output becomes **0** (Low/False), and vice versa. A triangle with a small circle at the output represents it.

The Boolean expression for a NOT gate is: $\bar{A}$ (bar) or A′ where A is the input.

Truth Table

A (Input)	Y = Ā (Output)
0	1
1	0

AND-OR LOGIC GATES

AND GATE

The **AND gate** is a digital logic gate that produces an output of **1 (True/High)** only when **both inputs are 1**; otherwise, the output is **0 (False/Low)**. It follows the multiplication rule, and if the inputs are **A** and **B**, the Boolean expression is **A • B**. This means the output is high only when all given conditions are satisfied

Examples

1. **Security System**: The alarm will sound **only if both** the **motion sensor** and **door sensor** detect activity (**1 AND 1 → 1**).
2. **Car Seatbelt Warning**: The seatbelt warning light turns **ON** only when the **engine is running** AND the **seatbelt is unfastened** (**1 AND 1 → 1**).

OR GATE

The **OR gate** is a digital logic gate that outputs **1 (True/High)** if **at least one of the inputs is 1**; otherwise, it outputs **0 (False/Low)**. It follows the addition rule, and if the inputs are **A** and **B**, the Boolean expression is **A + B**. This means the output is high when any one or both conditions are met.

Examples of OR Gate

1. **Automatic Streetlight**: The streetlight turns **ON** if **either the night sensor detects darkness OR a manual switch is pressed** (**1 OR 1 → 1**).
2. **Car Door Warning**: The dashboard warning light turns **ON** if **any car door is open**.
3. **Fire Alarm System**: The alarm will activate **if either smoke is detected OR a manual alarm button is pressed**.

2- Input AND Gate

Truth Table

A (Input 1)	B (Input 2)	X = (A.B)
0	0	0
0	1	0
1	0	0
1	1	1

2-Input OR Gate

Truth Table

Input A	Input B	X = A+B
0	0	0
0	1	1
1	0	1
1	1	1

UNIVERSAL LOGIC GATES (NAND, NOR)

- **Why are NAND and NOR Gates Called Universal Gates?**

Both NAND and NOR gates are called universal gates because: They can perform all basic logic operations (AND, OR, NOT, XOR, XNOR) without needing other gates. Any complex digital circuit can be designed using only NAND or only NOR gates

- **NAND Gate (NOT AND Gate)**

The **NAND gate** is a combination of an **AND** gate followed by a **NOT gate**. It outputs **0 (False/Low) only when both inputs are 1**, otherwise, it outputs **1 (True/High)**. This means it gives the opposite result of an AND gate. The Boolean expression is: $\overline{A.B}$ OR (A.B)'

Example:

Emergency Stop System: A machine stops only when both the safety buttons are pressed, otherwise, it keeps running.

Security Lock: A door remains **locked** unless both the **correct code is entered AND the fingerprint is verified** (inverting AND behavior).

NOR Gate (NOT OR Gate)

The **NOR gate** is a combination of an **OR gate followed by a NOT gate**. It outputs **1 (True/High) only when both inputs are 0**, otherwise, it outputs **0 (False/Low)**. It works as the **opposite of an OR gate**. The Boolean expression is: $\overline{A+B}$ OR (A+B)'

Example:

Streetlight Control: The light turns **ON** only when both daylight and motion sensors are **OFF** (**0 NOR 0 → 1**).

Buzzer System: A warning buzzer will stay silent unless both the temperature and pressure sensors are in safe (0) conditions

2-Input NAND Gate

Truth Table

Input A	Input B	X = (A.B)'
0	0	1
0	1	1
1	0	1
1	1	0

2- Input NOR Gate

Truth Table

Input A	Input B	0 = (A + B)'
0	0	1
0	1	0
1	0	0
1	1	0

Exclusive OR/NOR

XOR Gate (Exclusive OR Gate)

The XOR gate outputs 1 (True/High) only when the inputs are different; otherwise, it outputs 0 (False/Low). It follows the Boolean expression:

$A \oplus B = (A \cdot \bar{B}) + (\bar{A} \cdot B) = (A \cdot B') + (A' \cdot B)$

Example:

Parity Checker: Used in error detection, where it checks if the number of 1s in a binary string is odd or even.

Light Control System: A lamp connected to two switches turns ON when only one switch is flipped but stays OFF when both are the same.

XNOR Gate (Exclusive NOR Gate)

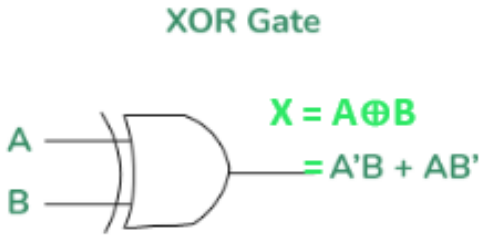
The XNOR gate is the inverse of the XOR gate. It outputs 1 (True/High) when the inputs are the same and 0 (False/Low) when the inputs are different. The Boolean expression is:

$A \odot B = \overline{A \oplus B} = (A \cdot B) + (\bar{A} \cdot \bar{B}) = (A \cdot B) + (A' \cdot B')$

Example:

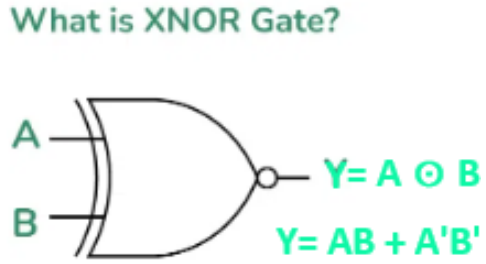
Digital Comparator: Used in circuits to check if two binary values are equal.

Authentication System: If a user enters the correct password, the system compares input with stored data and grants access only if they match (**1 XNOR 1 → 1**)



Truth Table

A (Input 1)	B (Input 2)	X = A'B + AB'
0	0	0
0	1	1
1	0	1
1	1	0



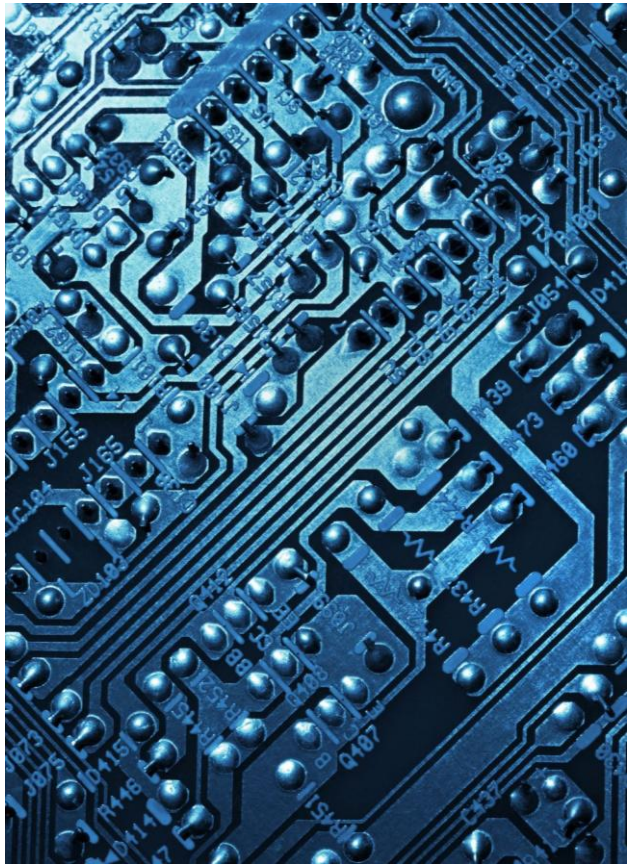
Truth Table

Input A	Input B	Output
0	0	1
0	1	0
1	0	0
1	1	1

Deterministic Computing

Deterministic computing refers to systems where the same input always produces the same output, ensuring predictability and consistency. It is used for tasks with fixed processes and no uncertainty, such as sorting data or designing digital circuits.

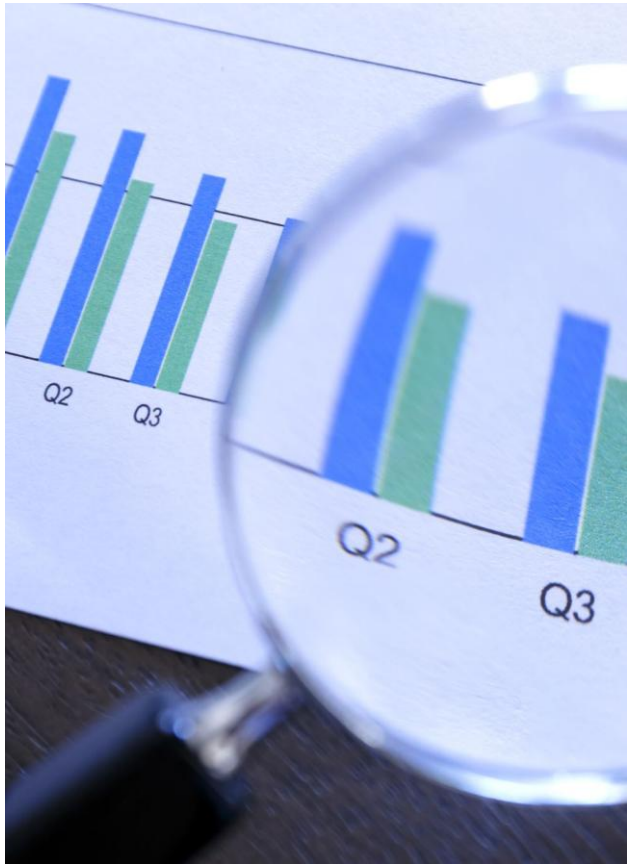
- Characteristics:
  - Consistency: Same inputs, same outputs.
  - Non-Adaptability: Fixed processes.
  - No Randomness: Predictable outcomes.
- Examples:
  - Sorting numbers, digital circuit design.
- Limitations:
  - Cannot handle uncertainty or dynamic environments.



Probabilistic Computing

Probabilistic computing involves systems that handle uncertainty and incomplete data by making predictions based on probabilities. It is adaptable and flexible, used in tasks like weather forecasting and fraud detection.

- Characteristics:
  -Handles uncertainty and incomplete data.
  - Uses statistical inference.
  - Adaptable and flexible.
- Examples:
  - Weather forecasting, fraud detection.



Deterministic vs Probabilistic Computing

• Deterministic Computing:	- Predictable and fixed.	- Example: Sorting, circuit design.
• Probabilistic Computing:	- Adapts to uncertainty.	- Example: Predictions, real-world problems.