
03 - Logic Gates and Circuits

Prof. Riccardo Berta

2025-09-10

Contents

1	Logic gates and circuits	1
1.1	Logic gates	1
1.1.1	NOT gate (Inverter)	2
1.1.2	BUFFER gate	2
1.1.3	AND gate	3
1.1.4	OR gate	3
1.1.5	XOR, NAND, NOR Gates	4
1.2	Boolean Algebra	5
1.3	Digital Circuits	6
1.3.1	Combinational circuits	8
1.3.2	Sequential circuits	8
1.3.3	Combinational composition	8
1.4	Complexity	9
1.5	Simulation	10
1.6	Exercises	11

1 Logic gates and circuits

Digital systems are built on the simplest possible foundation: two states, usually represented as 1 and 0. These two values are not just numbers; they can also be interpreted as **TRUE** and **FALSE**, **HIGH** and **LOW** voltage, or even as the **presence** or **absence** of a signal. This abstraction allows us to **ignore the messy details of electronics** and instead **reason in terms of logic**.

At the heart of digital design are **logic gates**, the elementary building blocks that take one or more binary inputs and produce a binary output according to a well-defined rule. By combining gates, we can implement any **logical function**, from simple comparisons to the complex operations inside a microprocessor. While the gates themselves are simple, the real power comes from combining them into larger modules that ultimately form the backbone of computers and digital devices.

1.1 Logic gates

Logic gates are **simple digital circuits** that accept one or more binary inputs and produce a binary output. Each logic gate is represented by a **symbol** that shows how inputs are connected to the output. By convention, the inputs are placed on the left (or sometimes at the top), while the output appears on the right (or at the bottom).

The relationship between the inputs and the output can be described in two equivalent ways:

- A **truth table** explicitly lists all possible input combinations in its left columns and the corresponding output values in the right column. In this way, it provides a complete specification of the gate's behavior.
- A **Boolean equation** is a mathematical expression that uses binary variables and logical operators to capture the same relationship in algebraic form.

Both descriptions are fundamental tools for analyzing and designing digital circuits.

1.1.1 NOT gate (Inverter)

A NOT gate is the simplest type of logic gate. It has one input, usually labeled A, and one output, usually labeled Y. The output is the inverse of the input:

- If A is FALSE (0), then Y is TRUE (1)
- If A is TRUE (1), then Y is FALSE (0)

The graphical symbol for the NOT gate is a triangle with a small circle (called a bubble) at its output, representing the inversion. The behavior can be summarized in the two complementary ways:

name: NOT symbol: $A \longrightarrow \text{triangle with bubble} \longrightarrow Y$ truth table:

A	Y
0	1
1	0

 boolean equation: $Y = \overline{A}$

The bar over the variable indicates the logical NOT operation, and is read as "Y equals NOT A". Because it reverses the logic value of its input, the NOT gate is also called an inverter.

1.1.2 BUFFER gate

The second one-input logic gate is the buffer. Unlike the NOT gate, the buffer **does not invert the signal**: it simply copies the input to the output:

- If the input A is 0, the output Y is also 0;
- If the input A is 1, the output Y is also 1.

This behavior can be expressed with the Boolean equation and the truth table:

name: BUFFER symbol: $A \longrightarrow \text{triangle} \longrightarrow Y$ truth table:

A	Y
0	0
1	1

 boolean equation: $Y = A$

From a **purely logical** point of view, the buffer seems redundant, since it performs the same function as a **direct wire connection** between input and output. However, from the **analog point of view**, the buffer provides important benefits. It is capable of delivering large amounts of current, for example to drive a motor, and it can rapidly distribute its output to many other gates in a circuit without signal degradation.


This example highlights the **importance of considering multiple levels of abstraction**. While the digital abstraction suggests that the buffer is unnecessary, at the physical implementation level it plays a critical role in ensuring reliable and efficient circuit operation.

1.1.3 AND gate

The AND gate is a **two-input logic gate** whose output is TRUE only when both inputs are TRUE. In every other case, the output is FALSE:

- If the inputs A is 1 and B is 1, the output Y is 1
- If one of the inputs A or B (or both) is 0, then the output Y is 0.

This behavior is summarized by the following truth table and Boolean equation:

name: AND	symbol:		truth table:	<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	0	1	0	0	1	1	1	boolean equation: $Y = AB$
A	B	Y																		
0	0	0																		
0	1	0																		
1	0	0																		
1	1	1																		

The equation can also be written in alternative forms, such as:

$$Y = AB$$

$$Y = A \cdot B$$

$$Y = A \cap B$$

The dot or simple juxtaposition are commonly used in digital design and read as "A AND B". The symbol cap, meaning intersection, is often preferred in logic and mathematics.

The AND gate is one of the most fundamental logic gates and is widely used to model situations where two or more conditions must be simultaneously satisfied.

1.1.4 OR gate

The OR gate is a logic gate that produces a TRUE output if at least one of its inputs is TRUE:

- If either input is 1 (or both are 1), the output will also be 1
- If both inputs are 0, the output will be 0.

name: OR



truth table:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

boolean equation: $Y = A+B$

The Boolean equation for the OR gate is written as:

$$Y = A + B \quad Y = A \cup B$$

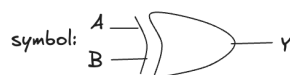
The plus sign is widely used by digital designers and is read as "A OR B". The union symbol is often preferred in mathematics and logic.

The OR gate is fundamental in digital systems because it models situations in which one condition, or another, or both can lead to a positive result.

1.1.5 XOR, NAND, NOR Gates

The AND, OR, and NOT gates are the fundamental logical operations. With them, we can construct all the digital operations we need. However, besides these basic operations, other logic gates exist. In some cases these gates are more convenient to use, or they allow the design of digital circuits to be simplified. Some of these gates are even **universal**: with them, it is possible to implement any logical function. Among the most common two-input gates we have the XOR (exclusive OR), pronounced "ex-OR", which is TRUE if A or B is true, but not both:

name: XOR



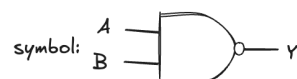
truth table:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

boolean equation: $Y = A \oplus B$

It is indicated by a plus sign with a circle around it; the NAND, which performs NOT AND. Its output is TRUE unless both inputs are true:

name: NAND



truth table:

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

boolean equation: $Y = \overline{AB}$

and the NOR, which performs NOT OR. Its output is TRUE only if neither A nor B is true:

name: NOR



truth table:

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

boolean equation: $Y = \overline{A+B}$

1.2 Boolean Algebra

Boolean algebra is the **mathematical framework** used to represent and manipulate logical expressions. It can be described as an **algebraic system**, which means it is a set of elements together with a set of operations defined on them. In this case, the system is defined by three components:

- the set of values $\{0, 1\}$,
- the operations OR, AND, and NOT,
- the equivalence operator $=$

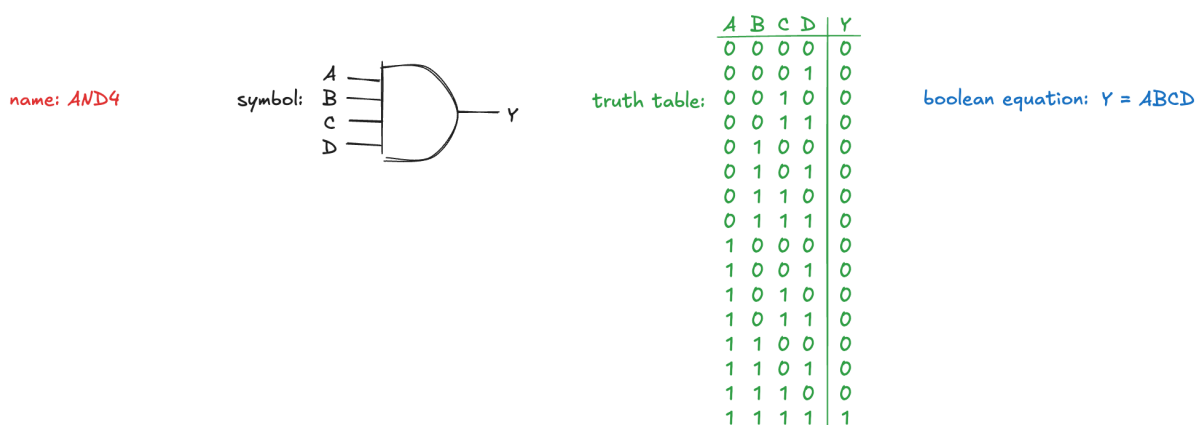
A **Boolean variable** is any discrete variable that can assume only two possible values, 0 or 1, and a **Boolean function** is any expression F that is formed using logical operations. It takes one or more binary inputs

(X_1, X_2, \dots, X_n)

and produces a binary output Y .

The **domain** of a Boolean function is made up of all the 2^n possible combinations of the values of the variables. This domain is **countable**, meaning that it **can be listed explicitly**, this means that a Boolean function can be completely specified by its Boolean expression or by a truth table that lists all possible input combinations and their corresponding output values.

Using the idea of Boolean function, we can extend the basic logic gates in order to handle multiple inputs. An **N-input** AND gate produces a TRUE output only when all N inputs are TRUE. For example, a 4-input AND gate (AND4) outputs 1 only when A, B, C, and D are all equal to 1:



An N-input OR gate produces a TRUE output when at least one input is TRUE. For example, a 3-input NOR gate outputs 1 only if all three inputs are 0:

name: **NOR3**

truth table:

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

boolean equation: $Y = \overline{A+B+C}$

These examples illustrate how basic gates can be generalized to handle more inputs, while still following the same logical definitions. Their behavior can always be described using symbols, Boolean equations, and truth tables.

1.3 Digital Circuits

A circuit is a **network** that processes Boolean variables. It can be viewed as a black box with one or more **input terminals** and one or **more output terminals**. To describe such a circuit, we use two kinds of specifications:

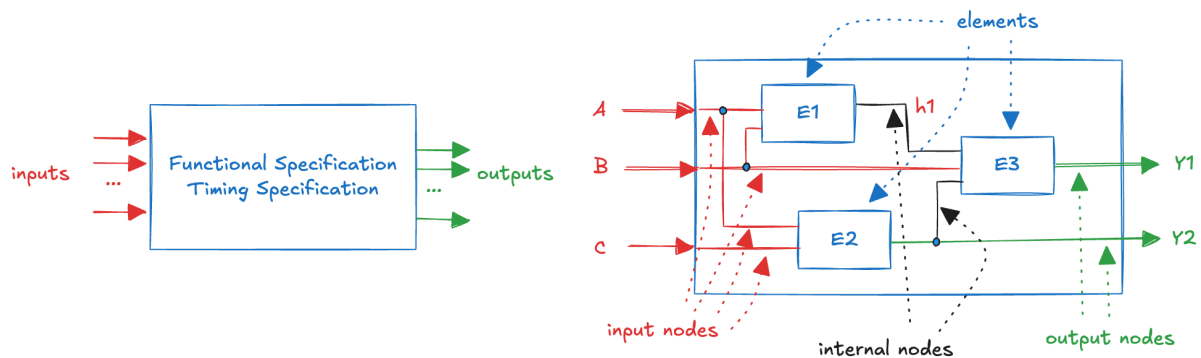
- a **functional specification**, which defines the **logical relationship between inputs and outputs**;
- a **timing specification**, which describes the **delay between a change in the inputs and the corresponding change in the outputs**.

Inside the black box, circuits are built from **nodes** and **elements**:

- elements are themselves circuits, each with inputs, outputs, and specifications.
- nodes are the wires that connect elements together, carrying a voltage that represents a Boolean variable.

There are three types of nodes:

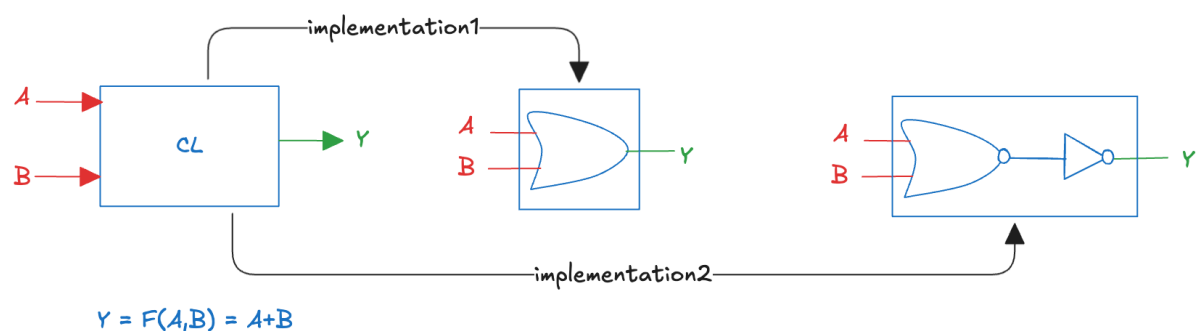
- **input nodes** receive values from the external world,
- **output nodes** deliver values to the external world,
- **internal nodes** carry signals inside the circuit but are not directly connected to the outside.



Sometimes, several nodes are **grouped together** to carry **related signals**. In this case, instead of drawing many separate wires, we use a **bus**, which is simply a bundle of multiple signals treated as a unit. A bus is represented by a single line with a slash and a number next to it, where the number indicates how many signals it contains. This compact notation makes circuit diagrams clearer and easier to read, especially when dealing with multi-bit values such as data words:



Together, nodes and elements form the **internal structure of digital circuits**, while from the outside the system can still be described simply by its inputs, outputs, and specifications. In practice, there are often **multiple possible implementations for the same logical function**:



The **choice** of implementation depends on the **available building blocks** and on **design constraints** such as area, speed, and power consumption. Different circuit realizations may compute the same Boolean function, but they can vary in terms of **efficiency** and **performance**.

Digital circuits can be divided into **combinational and sequential types**, depending on how

their outputs are determined.

1.3.1 Combinational circuits

A combinational circuit produces an output that **depends only on the current values of the inputs**. It simply combines the present input values to compute the output, without remembering anything from the past. For this reason, combinational circuits are said to be **memoryless**. A basic logic gate is an example of a combinational circuit.

1.3.2 Sequential circuits

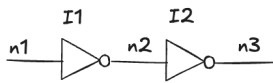
A sequential circuit produces an output that **depends on both the current and the previous values** of the inputs. In other words, the output is influenced by the sequence of inputs over time. Sequential circuits therefore have **memory**, and they can store information about past inputs.

1.3.3 Combinational composition

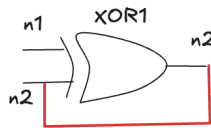
To design digital systems, we often want to build a **large combinational circuit** starting from smaller combinational circuit elements. This modular approach makes it easier to manage complexity and reuse standard building blocks. A circuit is combinational if it is composed of interconnected elements that satisfy the following conditions:

- every element is itself **combinational**,
- every node is **either an input** or is **connected to exactly one output** of an element,
- the circuit contains **no cyclic paths**, meaning that every path through the circuit visits each element at most once.

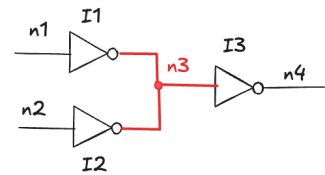
These rules are **sufficient** to guarantee that a circuit is combinational, but they are **not strictly necessary**. In fact, there exist circuits that violate one or more of these conditions and are still combinational. However, to simplify design and analysis, we **restrict ourselves** to combinational composition when building combinational circuits. Here some examples to clarify these concepts:



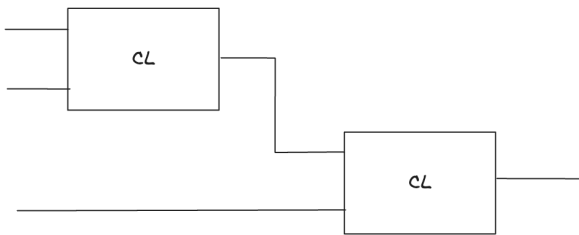
It is combinational circuit: the circuit is made of interconnected elements with no cyclic paths. Each node connects properly to a single output, so the rules are satisfied.



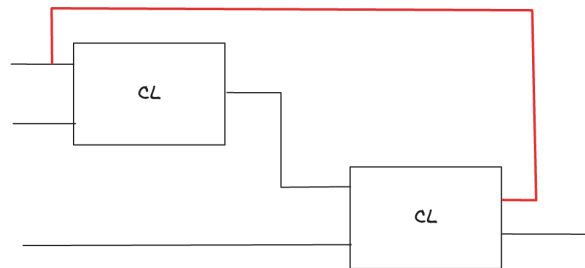
It is not combinational: there is a cyclic path in the circuit. This feedback loop prevents it from being purely combinational.



It is not combinational: node n3 is incorrectly connected to the outputs of two elements (I1 and I2). Since a node must connect to exactly one output, the rule is violated.



It is combinational: two combinational circuits are connected together to form a larger one. Since each sub-circuit follows the rules, the overall circuit is also combinational.



It does not obey the rules: the circuit contains a cyclic path through the two elements. Depending on the actual functions of these elements, the overall circuit may or may not be combinational, but it does not respect the strict rules of combinational composition.

1.4 Complexity

Large circuits, such as microprocessors, can be extremely complex. To manage this complexity, we rely on a few key principles:

- **Abstraction and modularity:** we can view a circuit as a black box with a well-defined interface and function, without worrying about the internal details.
- **Hierarchy:** we build complex systems by combining smaller circuit elements step by step, creating multiple levels of organization.
- **Discipline:** we apply the rules of combinational composition to ensure circuits behave correctly and remain easy to analyze.

Using these principles allows designers to break down a seemingly overwhelming problem into smaller, manageable parts.

The functional specification of a combinational circuit is usually expressed either as a truth table or as a Boolean equation. This raises some important design questions:

- How can we implement the basic gates AND, OR, and NOT?

- How can we derive a Boolean equation from any given truth table?
- How can we use Boolean algebra to simplify Boolean equations?

These questions guide the process of moving from abstract functional specifications to practical, efficient circuit implementations.

1.5 Simulation

Simulation is an essential step between theory and practice. When we design a circuit on paper, we rely on Boolean equations and truth tables to predict how it should behave. However, as circuits grow more complex, analyzing them manually becomes difficult and error-prone. Simulation bridges this gap by allowing us to **test our designs in a safe, controlled, and cost-free environment**. Before building any physical prototype, we can use simulation tools to:

- **Verify correctness:** check that the circuit produces the expected outputs for all possible inputs.
- **Explore behavior over time:** visualize how signals change, and how propagation delays affect the output.
- **Detect errors early:** find logical mistakes or design flaws before hardware is built, avoiding expensive rework.
- **Experiment freely:** try variations, optimize the design, and gain intuition about circuit behavior without worrying about damaging components.

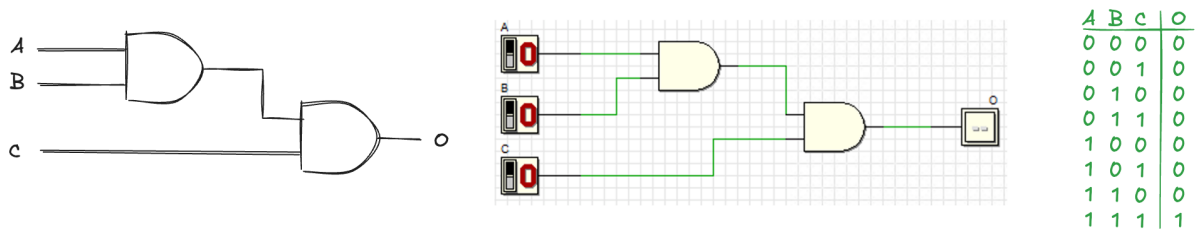
In education, simulation is even more valuable. It provides students with an interactive environment where they can directly connect theory to practice. By **building and testing circuits virtually**, learners see abstract concepts like logic gates, timing, and sequential behavior come to life. Ultimately, simulation is not just a convenience, it is a core methodology of modern digital design, enabling designers and students alike to move confidently from ideas to working systems. The first tool we will use is the **Deeds Digital Circuit Simulator**

DEEDS, short for **Digital Electronics Education and Design Suite**, is a comprehensive educational platform developed at the University of Genoa. It's designed not as a commercial tool but as an intuitive, hands-on environment for learning digital electronics. At the core of Deeds is the Digital Circuit Simulator, a tool tailored to help students design and interact with digital circuits. With its graphical schematic editor, it allows us to drag and drop components like logic gates then wire them intuitively on a canvas. What makes this tool uniquely pedagogical are its simulation modes:

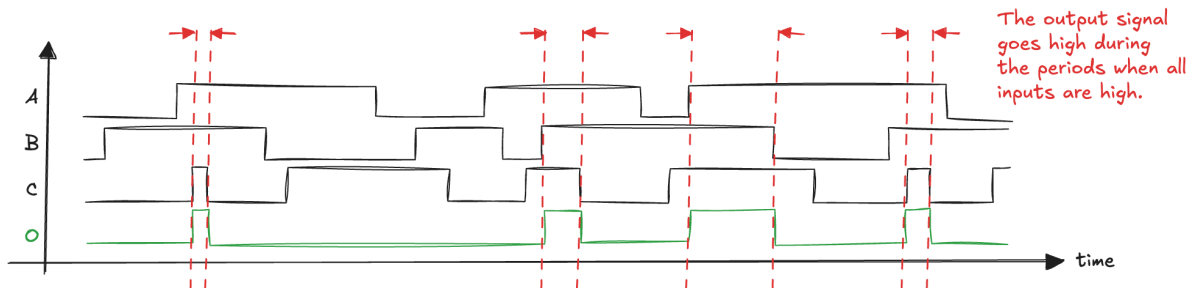
- the **interactive animation mode** lets us flip input switches on the schematic and observe real-time changes in outputs, which is ideal for exploring logic behavior dynamically.

- the **timing Simulation mode** offers a timing-diagram interface, where we can draw input signal sequences and analyze **output waveforms** step-by-step to understand temporal behavior.

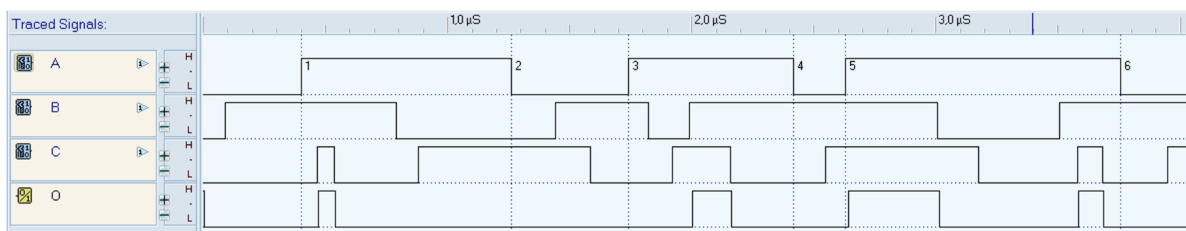
Together, these features bridge the gap between theoretical learning and practical understanding. Let's consider a practical example, we can implement a simple combinational circuit, create it in Deeds, simulate its behaviour and verify that it matches the expected truth table:



We can also examine how the circuit responds as the input signals change, using the simulator to generate the **output waveform** from the given input waveforms. For instance, in the circuit shown we obtain:



The Deeds simulator provides support for this type of analysis:



1.6 Exercises

1 - Draw the symbol, boolean equation, and truth table for a three-input OR gate, a three-input XOR gate, and a four-input XNOR gate

- three-input OR gate

name: OR3

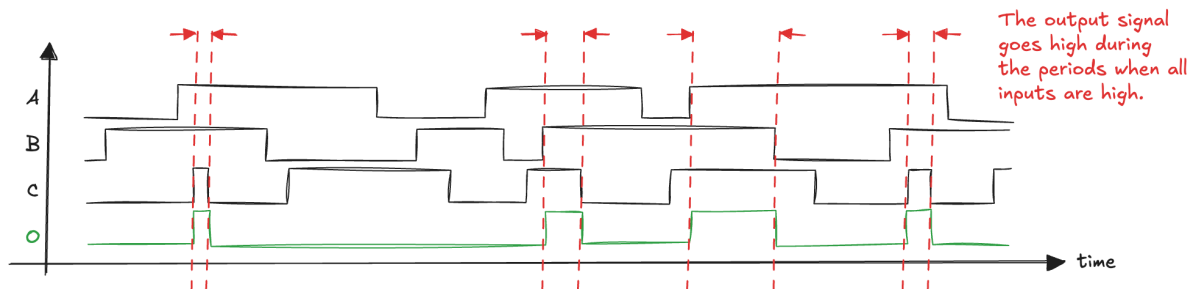


truth table:

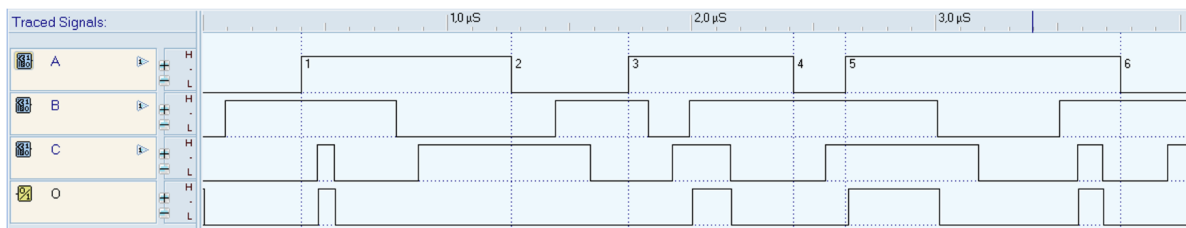
A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

boolean equation: $Y = A+B+C$

- three-input XOR gate



- four-input XNOR gate



2 - Draw the symbol, boolean equation, and truth table for a four-input OR gate, and a three-input XNOR gate

- four-input OR gate

name: OR4

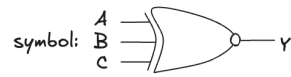


truth table:

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

boolean equation: $Y = A+B+C+D$

- three-input XNOR gate

name: **XNOR3**

truth table:

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

boolean equation: $Y = \overline{A \oplus B \oplus C}$

3 - A majority gate produces a TRUE output if and only if more than half of its inputs are TRUE. A three-input AND-OR gate produces a TRUE output if both A and B are TRUE or if C is TRUE. Complete a truth table for the two gates.

majority gate →

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

AND-OR gate →

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1