05 - Boolean Algebra

Prof. Riccardo Berta

2025-09-23

Contents

1 Boolean Algebra

Digital signals can take only a limited number of values. In the simplest and most important case, there are only two values: 0 and 1. To manipulate these binary variables, we use a mathematical framework called **Boolean Algebra**, introduced by the mathematician George Boole in the 19th century. The power of Boolean Algebra is that it allows us to **reason about digital circuits without worrying about their physical implementation**. A Boolean variable might be represented by a voltage (high = 1, low = 0), by the position of a mechanical switch, or even by fluid levels in a hydraulic system. From the point of view of Boolean Algebra, all these are equivalent. This abstraction is extremely useful for **hardware designers**, since we can design logic circuits by manipulating equations instead of analyzing transistors, and for **programmers**,

since software can be written using logic (true/false) without needing to understand the physical electronics. Later, we will see how Boolean expressions can be represented with logic gates (AND, OR, NOT, etc.) and how these gates form the basis of all digital systems, from simple calculators to modern microprocessors.

1.1 Fundamental Definitions

Before we can manipulate Boolean expressions or design digital circuits, we need to establish some precise terminology. Boolean algebra, like any branch of mathematics, is built upon a small set of well-defined concepts. These definitions will serve as the foundation for everything that follows, from truth tables to simplified logic circuits.

1.1.1 Variable

A Boolean variable is a **discrete variable** that can take only two possible values. By convention, we represent these values as:

- 0, which corresponds to False,
- 1, which corresponds to True.

This simple idea is extremely powerful because it allows us to **describe logical situations mathematically**. For example, a switch can be either off (0) or on (1), a digital signal can be low (0) or high (1), and a condition in a program can be false or true.

1.1.2 Function

Suppose we have a set of Boolean variables:

$X_1, X_2, \dots, X_n$

we can define a Boolean function over these variables, as a rule:

$f(X_1, X_2, \dots, X_n)$

that assigns a Boolean value (0 or 1) to every possible combination of its input variables:

- the function's output is always binary (either 0 or 1).
- its domain consists of all possible input combinations of the variables. Since each of the n variables can take only two values, the total number of combinations is 2^n.

For example, if we have two variables, there are $2^2 = 4$ possible combinations:

$(0,0), (0,1), (1,0), (1,1)$

Two Boolean functions are said to be **equivalent** if they produce the same output for every possible combination of inputs. Equivalence is important because **the same logical behavior can often be expressed in multiple ways**.

As an example, let's consider a simple Boolean function:

$f(X_1, X_2) = X_1 \cdot \overline{X_2}$

This means that the function is true if and only if X_1 is 1 and X_2 is 0. The corresponding truth table is:

X_1	X_2	$\overline{X_2}$	$f(X_1, X_2)$
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

This truth table shows clearly how the Boolean function works. For each of the $2^2 = 4$ possible input combinations, we calculate the output step by step.

1.1.3 Complement

Each variable can appear in two forms:

- **true form**, represented simply by the variable itself (e.g., A)
- **complementary form**, represented by a bar above the variable (e.g., $\overline{A}$)

The **complement** is the logical inverse:

- if $A = 0$, then $\overline{A} = 1$
- if $A = 1$, then $\overline{A} = 0$

This simple inversion is one of the fundamental operations of Boolean logic.

1.1.4 Literals

A literal is **an instance of a variable**, either in its true form or in its complemented form. For example, consider the Boolean function:

$f(A, B)$

This function contains two variables: A and B. If we define:

$f(A, B) = A + B$

this expression contains two literals: A and B, instead, if we define:

$f(A, B) = \overline{A}B + \overline{B}A$

this expression contains four literals: $\overline{A}$, B, $\overline{B}$, A.

It is important not to confuse variables with literals: each time a variable appears in an expression, it counts as a separate literal.

1.1.5 Products (Implicants) and Sums

The **AND** of one or more literals is called a **product** (or sometimes an **implicant**). Examples:

- AB,
- ABC,
- B.

All of these are considered products.

A **minterm** is a special kind of product, it **includes all the variables** of the function, each **appearing once** in either true or complemented form. For example, if we have three variables A, B, C:

- ABC,
- $A\overline{B}C$,
- $\overline{A}BC$,

are all minterms. Each minterm corresponds to exactly **one row of the truth table**.

The **OR** of one or more literals is called a **sum**. For example:

- $A + B$,
- $A + \overline{C}$,
- $A + B + C$.

are all sums.

A **maxterm** is a special kind of sum. It is a sum that **involves all the variables** of the function, each **appearing once** in either true or complemented form. For example, with three variables A, B, C:

$A + B + C$

is a maxterm. Just like minterms, each maxterm corresponds to exactly one row of the truth table, but in a different way, a minterm is true for one unique input combination, while a maxterm is false for one unique input combination

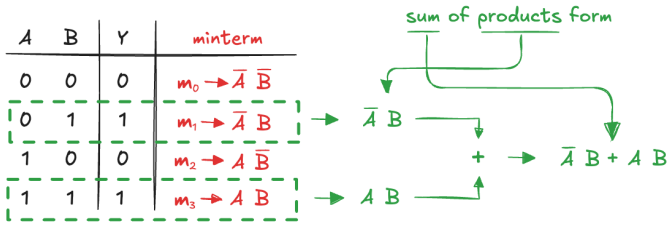
1.2 Canonical Forms

To represent Boolean functions in a systematic way from the truth table, we often use canonical forms. These are standard representations that express the output as a combination of all the input variables.

1.2.1 Sum-of-Products (SOP)

A Boolean function of N inputs can always be described by a truth table with 2^N rows, one for each possible combination of input values. To connect the truth table with an algebraic expression, we can exploit the minterm concept.

Let us look at an example with two input variables. The truth table has four rows, and thus four minterms:



To build a Boolean function from the truth table, we can take **the OR of all the minterms that correspond to rows where the output is equal to 1**. In the example, this means we take the minterms the second and fourth rows:

$m_1 = \overline{A}B$
$m_3 = AB$

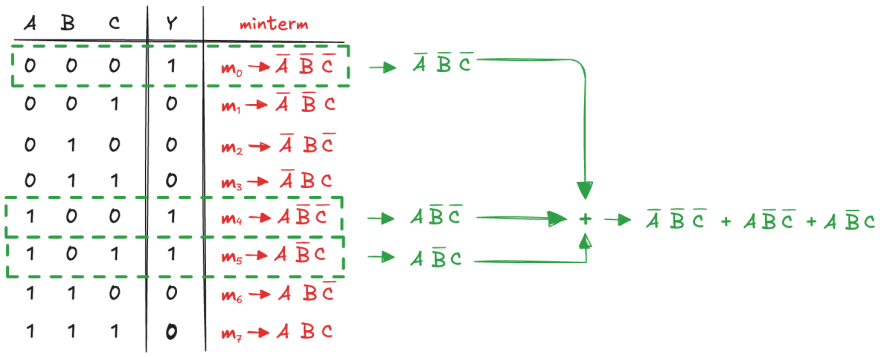
so the function is:

$Y = \overline{A}B + AB$

This expression is said to be in **Sum-of-Products (SOP) form**, because it is written as a sum (OR) of products (ANDs). Listing all of them explicitly can become lengthy, especially if the function has many inputs. To make expressions more compact, we can use the **sigma notation**. In this notation, a Boolean function is written as a sum of selected minterms, indicated by their index numbers:

$f(A, B) = \Sigma(m_1, m_3)$

The SOP form is important because it provides **a systematic way to write down a Boolean function directly from its truth table**. The method works not only for small functions but also for truth tables with any number of variables. Suppose we have three inputs, the truth table will contain $2^3 = 8$ rows:



Therefore, the Boolean function is:

$Y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$

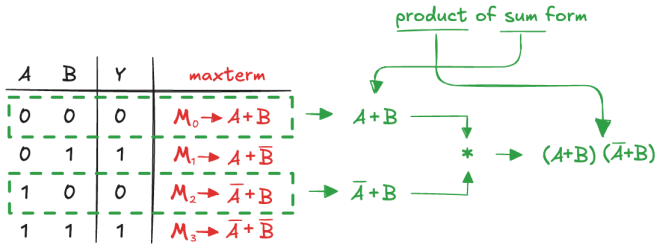
In sigma notation, we can write this compactly as:

$Y = \Sigma(m_0, m_4, m_5)$

Although the SOP form is systematic and always works, **it does not necessarily yield the simplest equation**. In the example above, the function uses three minterms, each of which contains three literals. With Boolean algebra simplification techniques, or with Karnaugh maps, we will later learn how to reduce this expression to a shorter and more efficient form.

1.2.2 Product-of-Sums (POS)

The Product-of-Sums (POS) form is an alternative way of expressing Boolean functions. Instead of using minterms (rows where the function is true), POS relies on **maxterms**, which are associated with rows where the function is false:



A Boolean function can be written as the AND of all the maxterms corresponding to the rows where the output is false. In the example, it means we take the maxterms for the first and third rows:

$Y = (A + B)(\overline{A} + B)$

This is the **Product-of-Sums (POS) form**, since it is a product (AND) of sums (ORs). In pi notation, we can write:

$f(A, B) = \Pi(M_0, M_2)$

Just like SOP, the POS form **does not necessarily yield the simplest possible expression**. However, the two forms are complementary:

- SOP is usually more convenient when the function is true for only a few rows.
- POS is often more convenient when the function is false for only a few rows.

This duality between SOP and POS is fundamental and will be useful when we learn simplification techniques.

1.3 Axioms and Theorems

Up to now, we have seen how any Boolean function can be written in a canonical form directly from its truth table. These methods are systematic and **guarantee a correct representation** of the function. However, the resulting **expressions are not always the most efficient**. A function written in canonical form may use many literals and therefore **require a large number of logic gates** when implemented.

1.3.1 Axioms

Boolean algebra is based on a small set of **axioms**, which we assume to be true. From these axioms, we can derive all other simplification rules. The table below lists five fundamental axioms that define the meaning of Boolean variables and the basic operations of NOT, AND, and OR:

Axiom	Statement	Dual Statement	Name
A1	$B = 0$ if $B \neq 1$	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	$\overline{1} = 0$	NOT
A3	$0 \cdot 0 = 0$	$1 + 1 = 1$	AND/OR
A4	$1 \cdot 1 = 1$	$0 + 0 = 0$	AND/OR
A5	$0 \cdot 1 = 1 \cdot 0 = 0$	$1 + 0 = 0 + 1 = 1$	AND/OR

A key property of Boolean algebra is the **principle of duality**. If in any valid equation we **systematically replace 0 with 1, 1 with 0, AND with OR, and OR with AND**, the new equation will also be **valid**. This principle allows us to generate pairs of related theorems, one being the dual of the other.

These axioms and the principle of duality form the foundation upon which all Boolean simplification techniques are built.

1.3.2 Theorems of one variable

The axioms form the foundation, but to make simplification practical we use a **set of theorems** derived from them. These theorems show how expressions can be reduced in a systematic way, saving gates and making circuits more efficient. Let us begin with the theorems involving a single variable. The following table summarizes five important theorems, together with their dual forms:

Theorem	Statement	Dual Statement	Name
T1	$B \cdot 1 = B$	$B + 0 = B$	Identity
T2	$B \cdot 0 = 0$	$B + 1 = 1$	Null element
T3	$B \cdot B = B$	$B + B = B$	Idempotency
T4	$\overline{\overline{B}} = B$	—	Involution
T5	$B \cdot \overline{B} = 0$	$B + \overline{B} = 1$	Complements

T1 - Identity Theorem: A variable ANDed with 1 is unchanged, and a variable ORed with 0 is also unchanged. In hardware terms, this means that if one input of an AND gate is permanently set to 1, the AND gate can be removed, and the output simply equals the other input:

And if one input of an OR gate is permanently set to 0, the OR gate can also be removed:

This is not only a theoretical simplification but also a **practical one**: **every unnecessary gate adds cost, power consumption, and delay to a circuit**. Replacing redundant gates with a direct wire connection is always beneficial.

T2 - Null element: A variable ANDed with 0 is always 0, and a variable ORed with 1 is always 1. This means that if one input of an AND gate is fixed to 0, the output will always be 0 regardless of the other input. In circuit terms, the gate is useless and can be replaced by a constant logic 0:

If one input of an OR gate is fixed to 1, the output will always be 1, no matter what the other input is. The OR gate can therefore be replaced by a constant logic 1:

These results are important for simplification because they allow us to eliminate redundant gates when one input is permanently tied to 0 or 1.

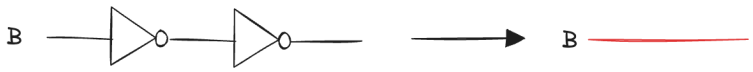
T3 - Idempotency: Repeating a variable does not change its value.

The word ”idempotent” comes from Latin roots meaning ”the same power”. In Boolean algebra, it refers to the fact that repeating the same variable in an operation has no additional effect. From a circuit perspective, this means that if both inputs of a gate are connected to the same signal, the gate is unnecessary: the output is identical to the input. The gate can be removed and replaced with a direct wire connection:

This theorem reinforces the idea that gates should never be used to duplicate an input; redundancy does not change the logic and only wastes hardware resources.

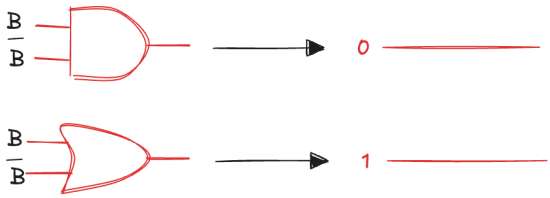
T4 - Involution: The complement of the complement returns the original variable.

This property is intuitive: the first negation flips the value, and the second negation flips it back. From a circuit perspective, two inverters connected in series cancel each other out and are logically equivalent to a direct wire connection:



T5 - Complements: A variable ANDed with its complement is always 0, while a variable ORed with its complement is always 1.

The complement theorem expresses the fundamental relationship between a variable and its logical opposite It is true because one of the two inputs must be 0 (for AND) or 1 (for OR). At the hardware level, they show that combining a signal with its inverted form leads to trivial constants (always 0 or always 1). Such expressions can be replaced by fixed values, reducing unnecessary gates:



1.3.3 Theorems of several variables

While the theorems of one variable provide useful simplifications, most practical Boolean expressions involve several variables. In this case, additional theorems allow us to reorganize and reduce expressions, often leading to much simpler forms. The table below summarizes some of the most important multi-variable theorems, together with their duals:

Theorem	Statement	Dual Statement	Name
T6	$B \cdot C = C \cdot B$	$B + C = C + B$	Commutativity
T7	$(B \cdot C) \cdot D = B \cdot (C \cdot D)$	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \cdot C) + (B \cdot D) = B \cdot (C + D)$	$(B + C) \cdot (B + D) = B + (C \cdot D)$	Distributivity

Theorem	Statement	Dual Statement	Name
T9	$B + (B \cdot C) = B$	$B \cdot (B + C) = B$	Covering
T10	$(B \cdot C) + (B \cdot \overline{C}) = B$	$(B + C) \cdot (B + \overline{C}) = B$	Combining
T11	$(B \cdot C) + (\overline{B} \cdot D) + (C \cdot D) = (B \cdot C) + (\overline{B} \cdot D)$	$(B + C) \cdot (\overline{B} + D) \cdot (C + D) = (B + C) \cdot (\overline{B} + D)$	Consensus
T12	$A \cdot B + \overline{A} = B + \overline{A}$	$(A + B) \cdot \overline{A} = B \cdot \overline{A}$	Absorption
T13	$B_0 \cdot B_1 \cdot B_2 \cdots = (\overline{B_0} + \overline{B_1} + \overline{B_2} \dots)$	$B_0 + B_1 + B_2 \cdots = (\overline{B_0} \cdot \overline{B_1} \cdot \overline{B_2} \dots)$	De Morgan's Theorem

Together, these theorems give us a powerful toolkit for manipulating Boolean functions into simpler and more efficient forms.

T6 — The Commutativity Theorem

This theorem shows that the **order of the inputs does not matter**, either for AND or for OR.

$B \cdot C = C \cdot B$
$B + C = C + B$

This property lets us **rearrange terms** without changing the logic of the expression, which is often a first step in aligning terms for further simplification.

T7 — The Associativity Theorem

This theorem shows that the **grouping of inputs does not matter**, either for AND or for OR. Parentheses can be rearranged freely:

$(B \cdot C) \cdot D = B \cdot (C \cdot D)$
$(B + C) + D = B + (C + D)$

Associativity lets us simplify expressions like A+B+C **without worrying about how the terms are grouped**. This makes Boolean expressions more flexible to handle.

T8 — The Distributivity Theorem

This theorem shows how AND and OR interact: **AND distributes over OR**, and **OR distributes over AND**:

$(B \cdot C) + (B \cdot D) = B \cdot (C + D)$
$(B + C) \cdot (B + D) = B + (C \cdot D)$

Notice that this property is **different from traditional algebra**, because in ordinary arithmetic addition does not distribute over multiplication. In Boolean algebra, however, the restricted values (0 and 1) make this distribution possible. This theorem is particularly important because it allows us to **factor expressions** (like algebraic factoring) or to **expand them**. Both directions are widely used in simplification.

T9 — The Covering Theorem

This theorem states that when a variable already appears in an expression, any additional term where the same variable is ANDed with another factor is redundant and can be eliminated:

$B + (B \cdot C) = B$
$B \cdot (B + C) = B$

This result is important since it allows us to **recognize and remove redundant terms**. We can prove this theorem using some of the previous theorems:

$B \cdot (B + C) = $ (T8 Distributivity)
$B \cdot B + B \cdot C = $ (T3 Idempotency)
$B + B \cdot C = $ (T8 Distributivity)
$B \cdot (1 + C) = $ (T2 Null element)
$B \cdot 1 = $ (T1 Identity)
B

T10 — The Combining Theorem

This theorem states that when two terms differ only by a variable and its complement, the **expression can be simplified to depend solely on the common factor**:

$(B \cdot C) + (B \cdot \overline{C}) = B$
$(B + C) \cdot (B + \overline{C}) = B$

This result is important since it allows us to eliminate variables that do not affect the outcome. We can prove it:

$(B \cdot C) + (B \cdot \overline{C}) =$ (T8 Distributivity)
$B \cdot (C + \overline{C}) =$ (T5 Complements)
$B \cdot 1 =$ (T1 Identity)
B

T11 — The Consensus Theorem

The theorem states that when two terms already cover all possible cases of a third term, that third term is **redundant** and can be removed without affecting the result:

$(B \cdot C) + (\overline{B} \cdot D) + (C \cdot D) = (B \cdot C) + (\overline{B} \cdot D)$
$(B + C) \cdot (\overline{B} + D) \cdot (C + D) = (B + C) \cdot (\overline{B} + D)$

Eliminating the redundant term reduces the number of gates without changing the function. We can prove it:

$(B \cdot C) + (\overline{B} \cdot D) + (C \cdot D) =$ (T5 Complements)
$(B \cdot C) + (\overline{B} \cdot D) + (C \cdot D \cdot (B + \overline{B})) =$ (T8 Distributivity)
$(B \cdot C) + (\overline{B} \cdot D) + (B \cdot C \cdot D) + (\overline{B} \cdot C \cdot D) =$ (T6 Commutativity)
$(B \cdot C) + (B \cdot C \cdot D) + (\overline{B} \cdot D) + (\overline{B} \cdot C \cdot D) =$ (T8 Distributivity)
$B \cdot C \cdot (1 + D) + \overline{B} \cdot D \cdot (1 + C) =$ (T2 Null element)
$B \cdot C \cdot 1 + \overline{B} \cdot D \cdot 1 =$ (T1 Identity)
$(B \cdot C) + (\overline{B} \cdot D)$

T13 — The Absorption Theorem

This theorem states that when a variable is combined with a term that already includes its complement, the redundant factor can be absorbed, leaving a simpler equivalent expression:

$A \cdot B + \overline{A} = B + \overline{A}$
$(A + B) \cdot \overline{A} = B \cdot \overline{A}$

We can prove it:

\$A\cdot B+\overline{A} = \$ (T5 Complements)
$A \cdot B + \overline{A} \cdot (B + \overline{B}) =$ (T8 Distributivity)
$A \cdot B + \overline{A} \cdot B + \overline{A} \cdot \overline{B} =$ (T8 Distributivity)
$B \cdot (A + \overline{A}) + \overline{A} \cdot \overline{B} =$ (T5 Complements)
$B \cdot 1 + \overline{A} \cdot \overline{B} =$ (T1 Identity)
$B + \overline{A} \cdot \overline{B} =$ (T8 Distributivity)
$(B + \overline{A}) \cdot (B + \overline{B}) =$ (T5 Complements)
$(B + \overline{A}) \cdot 1 =$ (T1 Identity)
$B + \overline{A}$

T13 — De Morgan's Theorems

De Morgan's theorems, named after the British mathematician Augustus De Morgan (1806–1871), who contributed significantly to logic and algebra, describe how the complement (negation) of a product or a sum can be expressed in terms of the complements of the individual terms.

Augustus De Morgan, 1806 - 1871

A British mathematician, born in India. Blind in one eye. His father died when he was 10. Attended Trinity College, Cambridge, at age 16, and was appointed Professor of Mathematics at the newly founded London University at age 22. Wrote widely on many mathematical subjects, including logic, algebra, and paradoxes. De Morgan's crater on the moon is named for him. He proposed a riddle for the year of his birth: "I was x years of age in the year x^2."

There are two fundamental forms: the complement of a product is equal to the sum of the complements:

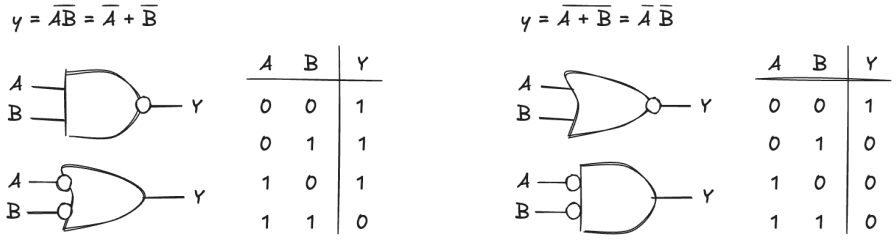
$$\overline{B \cdot C \cdot D} = \overline{B} + \overline{C} + \overline{D}$$

The complement of a sum is equal to the product of the complements:

$$\overline{B + C + D} = \overline{B} \cdot \overline{C} \cdot \overline{D}$$

In words: **the negation of an AND operation is equivalent to an OR of the negated inputs**, and **the negation of an OR operation is equivalent to an AND of the negated inputs**.

De Morgan's theorems are particularly important because they explain the **relationship between NAND/NOR gates and the basic AND/OR operations**. A NAND gate can be seen as an OR gate with inverted inputs, and a NOR gate can be seen as an AND gate with inverted inputs:



This equivalence allows NAND and NOR gates to be used as **universal gates**, since any digital circuit can be built using only NAND or only NOR gates.

The inversion circle drawn at the input or output of a gate is called a **bubble**. De Morgan's theorem explains how "pushing" a bubble through a gate changes its type: an AND gate becomes an OR gate when bubbles are moved to its inputs, and an OR gate becomes an AND gate when bubbles are moved to its inputs. This "bubble pushing" technique is widely used in circuit simplification and in schematic transformations, because it allows the designer to switch between AND/OR implementations using NAND and NOR gates.

1.3.4 Perfect induction

When working with Boolean algebra, many theorems can be proven algebraically using the axioms and previously established theorems. However, since Boolean variables can only take the values 0 or 1, there is another straightforward method available: perfect induction.

Perfect induction consists of **checking that a theorem holds for all possible input combinations of the variables involved**. If the two sides of the equation produce identical results for every case, then the theorem is proven. This method is direct, mechanical, and especially useful for verifying the correctness of Boolean laws. For example, we can use perfect induction to prove the consensus theorem. We construct a truth table for both sides of the equation:

B	C	D	$BC+\bar{B}D+CD$	$BC+\bar{B}D$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

Since the two columns are identical, the equality holds for all possible values of the inputs. Therefore, the theorem is proved.

Perfect Induction is **simple to understand**, since it only requires evaluating truth tables, and it is **guaranteed to work**, because Boolean variables are **finite** and **discrete**. However, it **becomes impractical** for expressions with many variables, because the number of rows grows as 2^n. In such cases, algebraic simplification is preferred.

1.4 Simplying Expressions

The theorems of Boolean algebra are not only useful for proving properties, but also provide a **systematic way to simplify Boolean expressions**. Simplification is at the heart of digital design. It transforms truth-table derived SOP or POS forms, which are often large and unwieldy, into compact implementations that use the fewest possible gates.

Whenever we apply Boolean theorems to reduce an expression, the simplified result remains logically equivalent to the original. One may then ask: why bother with simplification if the logical behavior does not change? The main reason is **efficiency**. A simplified expression generally requires **fewer logic gates to implement**. Since each gate consumes physical resources

in a circuit, fewer gates mean the design is **smaller**, **cheaper**, and **faster**, while also **consuming less power**. Simplification is therefore not just a theoretical exercise, but a fundamental step in optimizing digital hardware.Considering the following example:

$Y = \overline{A}B + AB$

At first glance, this may look like it requires two product terms. However, by applying the Combining Theorem (T10), we can simplify it directly to:

$Y = B$

The example shows how an apparently more complex expression can collapse into a very simple form when the right theorems are applied. In practice, more complicated functions may require several steps of simplification, but the principle remains the same.

The essential strategy for simplify an expression in SOP form is to repeatedly look for pairs of terms that differ in only one literal and use the combining rule to eliminate redundancy:

$PA + \overline{P}A = A$

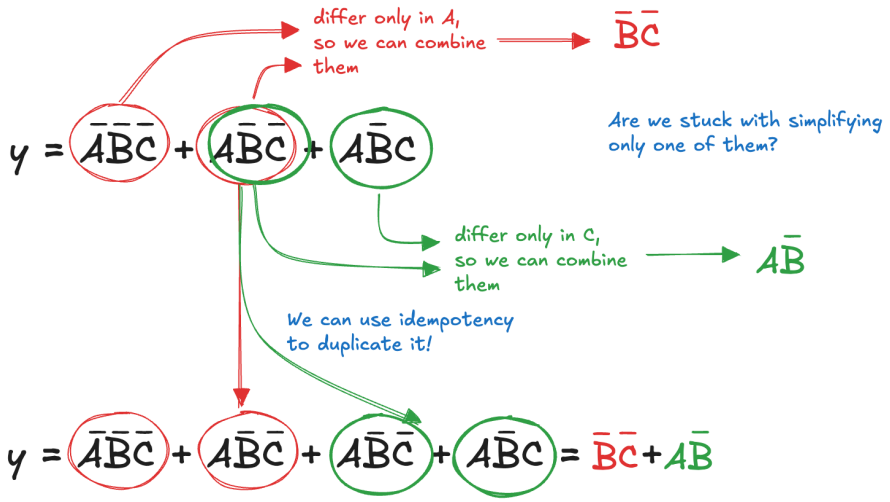
where P may represent any Boolean expression. This identity shows that if A appears in both terms, the value of P or its complement becomes irrelevant, and the function depends only on A.

How far can we go with simplification? An expression written in sum-of-products form is considered minimized if it uses the **fewest possible product terms**. When several different expressions use the same number of products, the minimal one is the one containing the **fewest total literals**. This criterion ensures that the minimized form corresponds to the most efficient implementation of the function.

In this context, a product is called a **prime product** if it cannot be combined with any other product in the expression to form a new product with fewer literals. Every product in a minimal equation must be a prime product. If a product is not prime, it can be merged with another term, which means the expression is not yet fully simplified.

Let us minimize the following expression:

$f = \overline{A}\,\overline{B}\,C + A\,\overline{B}\,C + A\,\overline{B}\,\overline{C}$



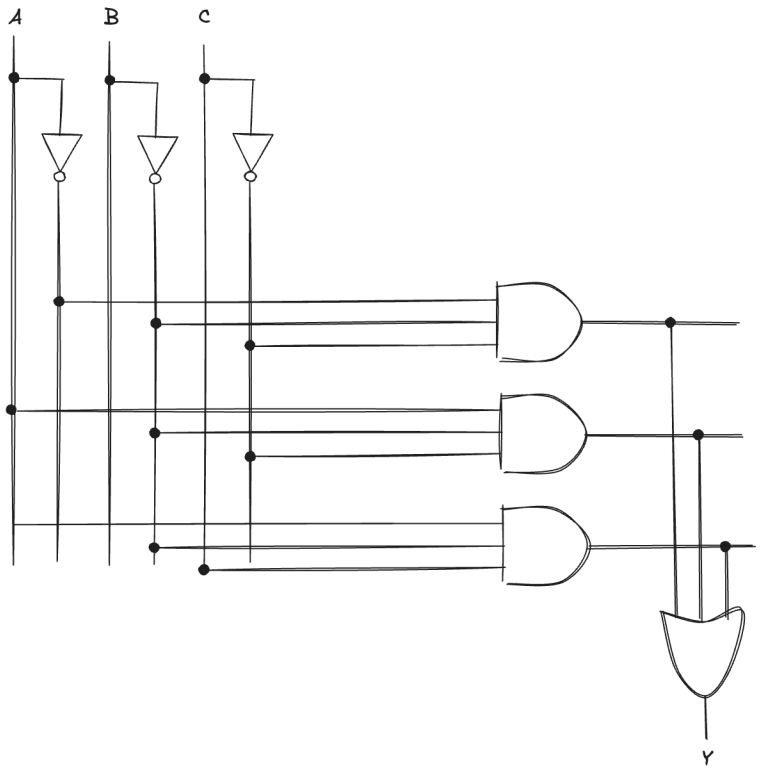
Simplifying Boolean expressions using only theorems can **sometimes be a matter of trial and error**. Different combinations of theorems may need to be tested before the minimal form emerges, and **the process can become complicated as the number of variables grows**. For this reason, in addition to algebraic manipulation, designers also rely on a graphical method known as the **Karnaugh map**, that provides a structured, visual way to identify simplifications, making the process faster and more reliable than relying solely on theorem application.

1.5 Schematic

A Boolean expression can be directly implemented as a digital circuit, and the resulting diagram is called a **schematic**. It shows the logic gates that realize the function and the wires that connect them. Consider the function of the previous exmaple:

$$Y = \overline{A}\,\overline{B}\,C + A\,\overline{B}\,C + A\,\overline{B}\,\overline{C}$$

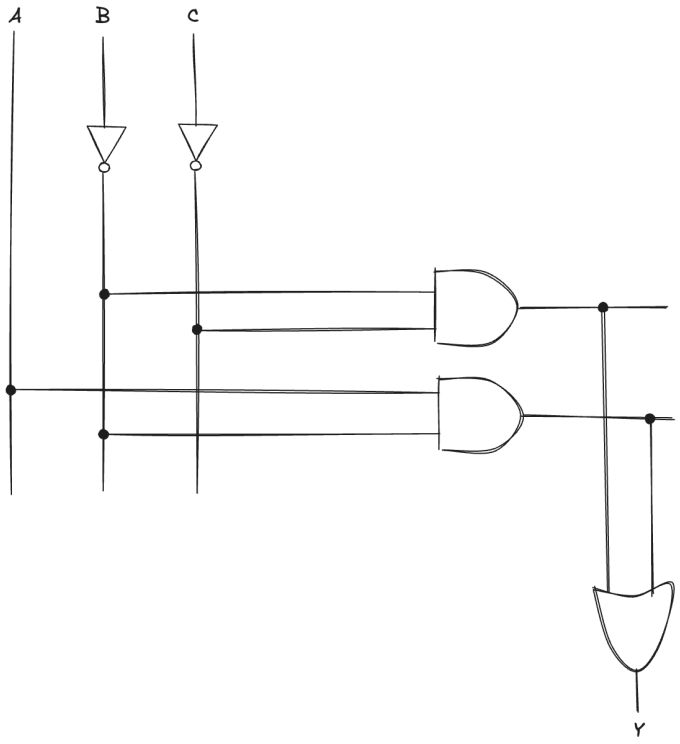
If we implement this function in its canonical sum-of-products form, the schematic requires three AND gates to generate the minterms and one OR gate to combine them, plus inverters for the complemented inputs. The circuit is correct, but it uses many gates and several inputs per gate:



After simplification, the same function reduces to

$Y = \overline{B}\,\overline{C} + A\,\overline{B}$

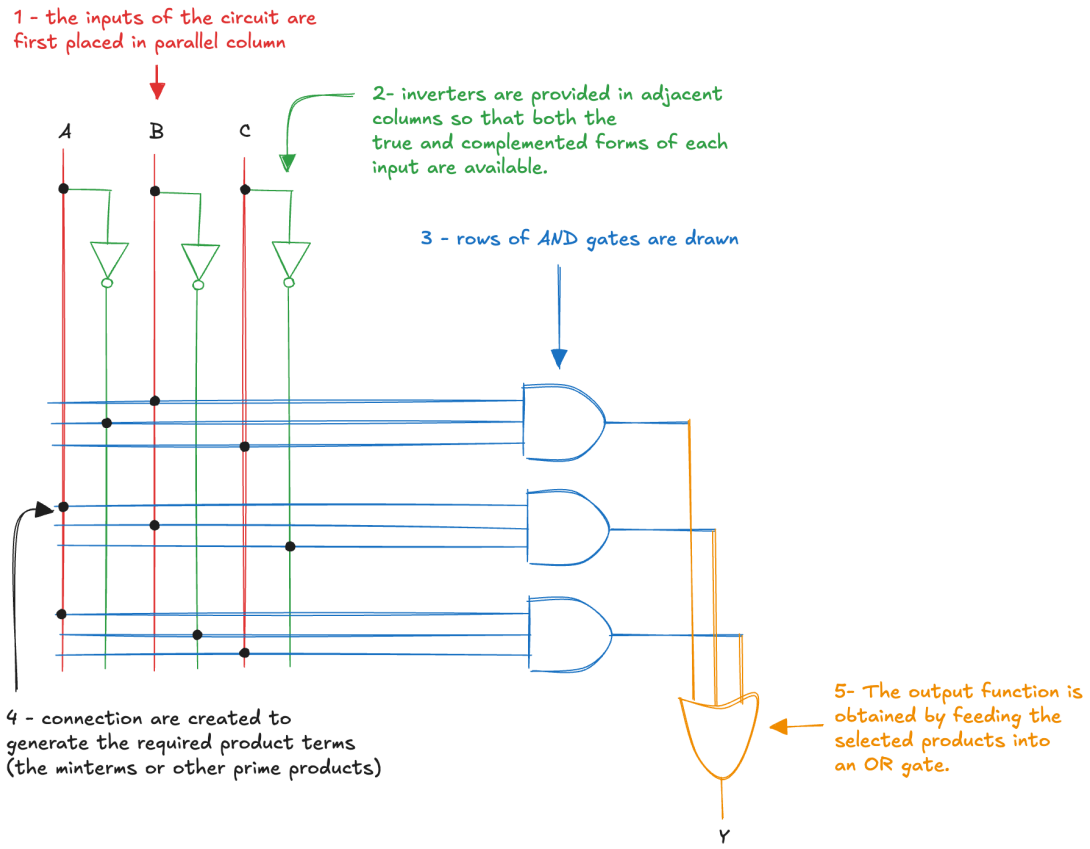
The corresponding schematic is much smaller: only two AND gates and one OR gate are required, along with a single inverter:



This comparison illustrates the central motivation for simplification. Although both circuits implement exactly the same logic, the simplified version requires** significantly less hardware**. Fewer gates mean a** smaller circuit**, which is generally** cheaper** and more **energy efficient**, and may also be **faster** because each signal passes through fewer gate inputs and fewer logic levels. This demonstrates how mathematical manipulation directly influences the cost, size, and performance of real circuits.

1.5.1 Programmable Logic Arrays (PLAs)

Notice from the previous example, that a Boolean equation written in sum-of-products form can be implemented in hardware following a **systematic procedure**:



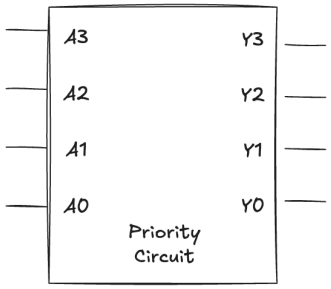
This regular structure of inputs, inverters, AND gates, and OR gates is called a **Programmable Logic Array (PLA)**. The name comes from the fact that the array of gates is arranged in a standard and systematic way, while **the actual logic function implemented depends only on how the connections** ("programming") are made between inputs, products, and outputs.

The advantage of the PLA style is that it provides a **universal template** for implementing any Boolean function. Instead of designing a new schematic from scratch for each function, we can always rely on the same architecture: a layer of input signals with inverters, a layer of AND gates forming products, and a layer of OR gates producing the outputs. The programming step determines which minterms are used and how they are combined.

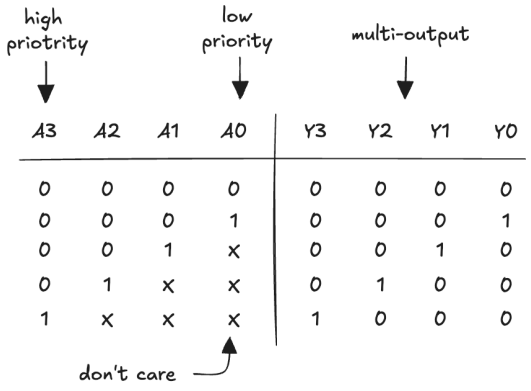
1.5.2 Multiple-output circuits

In many practical situations, digital circuits are required to generate **more than one output**. Each output is a separate Boolean function of the same set of inputs. To describe such circuits, we can either write an independent truth table for each output, or, more efficiently, place all the outputs in a single truth table that lists every input combination along with the corresponding values of all outputs. For example, we can consider a **priority circuit**, in which multiple inputs

may be active at the same time, but only the highest-priority active input is recognized and produces the corresponding output:

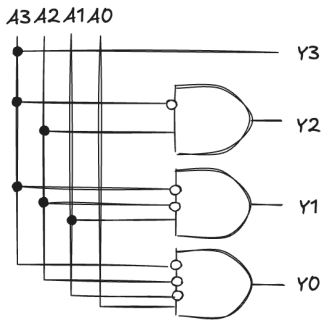


We can derive the Boolean equations for each output by constructing the canonical sum-of-products form and then simplifying with Boolean algebra. However, in practice, experienced designers often take advantage of observations and implement circuits ***directly from inspection***, bypassing lengthy algebraic simplifications. In this example, the outputs are true if the corresponding input is true and all higher-priority inputs are false. This leads directly to the following simplified thruth table:



A3	A2	A1	A0	Y3	Y2	Y1	Y0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

The X symbol indicates **"don't care" conditions**, meaning that the output can be either 0 or 1 without affecting the circuit's correctness. This flexibility allows for further simplification of the Boolean expressions. The resulting circuit is:



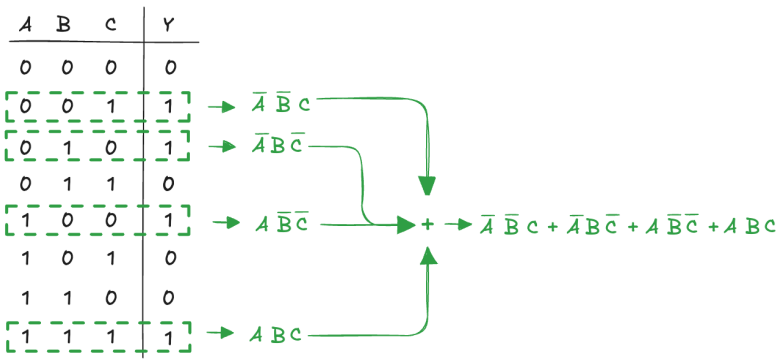
The schematic for the priority circuit shows how each output is generated with a small number of gates, reflecting the priority structure encoded in the truth table.

1.5.3 Multi-level circuits

Logic functions expressed in sum-of-products form are known as **two-level logic**, since they consist of literals connected to a layer of AND gates, whose outputs are then fed into a single OR gate. This representation is systematic and always correct, but for many functions it may require **an enormous amount of hardware**, as every minterm must be explicitly generated and summed.

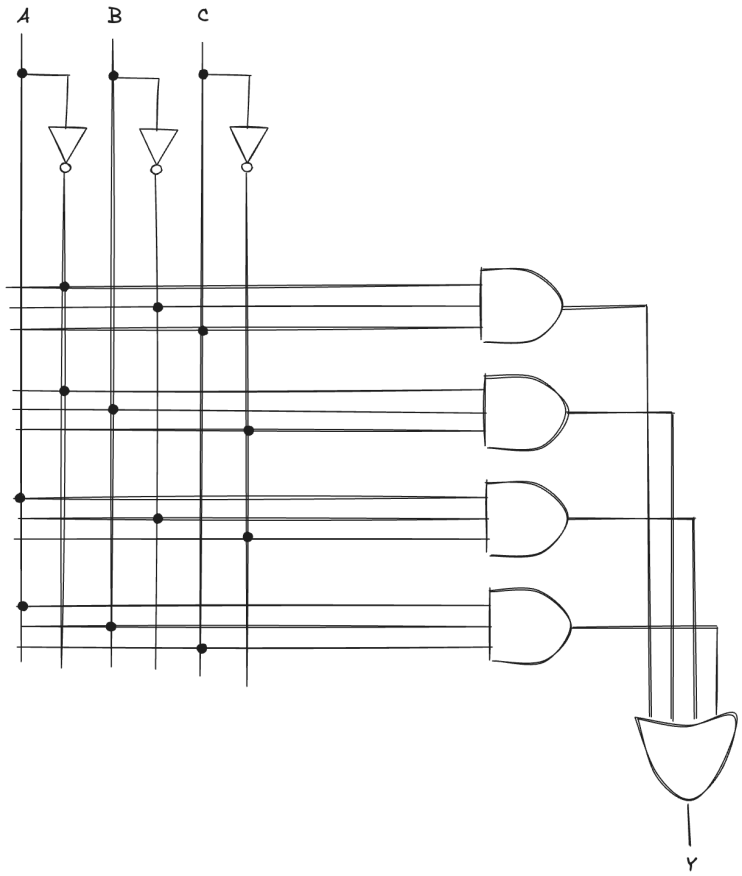
A useful example is the XOR function with three inputs. By definition, an XOR produces a 1 whenever an odd number of inputs are true. Looking at the truth table of:

$y = A \oplus B \oplus C$



The corresponding sum-of-products expression is:

$Y = A\overline{B}\overline{C} + \overline{A}B\overline{C} + \overline{A}\overline{B}C + ABC$



This implementation is correct but requires several AND gates and a large OR gate. A more compact solution emerges if we rewrite the function as

$A \oplus B \oplus C = (A \oplus B) \oplus C$

In this form, the circuit can be built simply by cascading two XOR gates, which is far more efficient than the canonical two-level approach.

This example illustrates the essence of **multilevel combinational logic**: by reorganizing Boolean expressions into multiple levels, we can obtain simpler circuits, with fewer gates, reduced hardware cost compared to their two-level counterparts.

However, choosing the **best** multilevel realization of a logic function is not straightforward. The notion of "best" depends on many different criteria: it might mean the circuit with the fewest gates, the fastest response, the shortest design time, the lowest cost, or the least power consumption. Importantly, the best design in one technology may not be the best in another. For example, CMOS circuits generally favor NAND and NOR gates rather than AND and OR gates, because they are simpler and more efficient to implement.

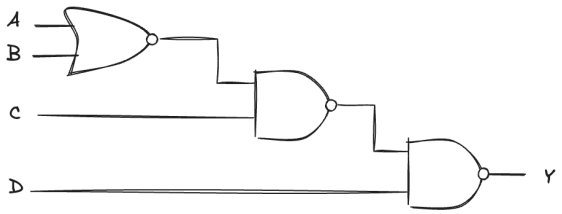
With growing experience, many circuits can be designed efficiently by inspection, as patterns

and simplifications become familiar. Nevertheless, for complex logic functions, the number of possible multilevel implementations grows rapidly, and **Computer-Aided Design** (**CAD**) tools are essential. These tools can explore a vast space of alternatives and automatically search for the design that satisfies given constraints, using the available building blocks.

1.5.4 Bubble pushing

When working with implementations based on NAND and NOR gates, it is often useful to redraw a circuit in a form where the underlying function can be determined more easily. A systematic method for this simplification is known as **bubble pushing**. The idea is to manipulate the "bubbles" (inversion markers) on the inputs and outputs of gates so that they cancel each other whenever possible, thereby revealing a clearer and more compact representation of the circuit.

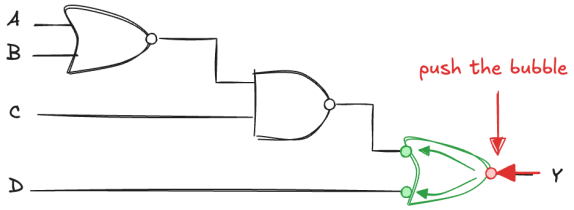
The procedure begins by examining the output of the circuit and then working backward toward the inputs. At each step, **inversion bubbles are pushed back**, using the property that a bubble at the input of one gate can be transferred to the output of the previous gate, provided the corresponding inversion is preserved. In practice, this means we can redraw each gate in such a way that input and output bubbles cancel when they appear in series. For example, consider the following circuit:

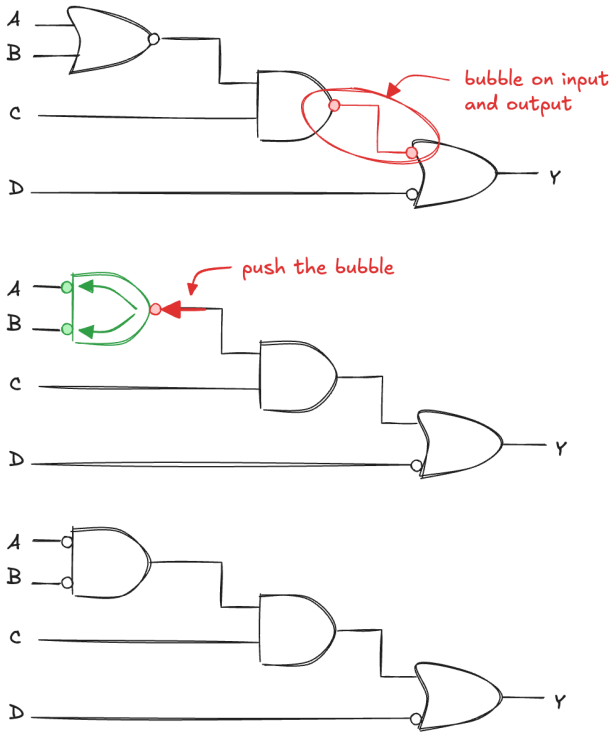


It's direct expression is:

$$Y = \overline{((\overline{A+B})\,C)D}$$

which is correct but not very transparent. By pushing the bubbles back through the gates, we can redraw the circuit step by step:





Finally, the circuit expression becomes:

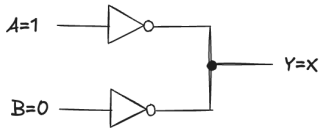
$Y = \overline{A}\,\overline{B}\,C + \overline{D}$

This technique allows us to read expressions directly in terms of the original signals rather than their complements, and it greatly simplifies circuits that contain large numbers of NAND and NOR gates. The final result is a representation that is both easier to analyze.

1.5.5 Beyond 0 and 1: X and Z

Boolean algebra strictly assumes only two logic levels, 0 and 1. Real circuits, however, may also exhibit **illegal or floating values**, usually denoted as **X** and **Z**. These values reflect physical situations that cannot be represented by ideal Boolean logic but occur frequently in hardware.

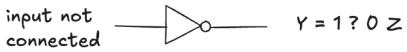
The symbol X indicates that a circuit node has an unknown or illegal value. This typically happens when the node is simultaneously driven to 0 and to 1, a situation known as **contention**:



In such a case the actual voltage lies somewhere between ground and power supply, depending on the relative strength of the driving sources. This is an **error condition** that must be avoided

in real systems. Circuit simulators also use X to represent an **uninitialized node**. It is important to remember that in truth tables X can also mean "don't care", but this is a different meaning that should not be confused with illegal values in circuits.

The symbol Z indicates that a node is being driven by neither a logic HIGH nor a logic LOW; in other words, the node is floating.

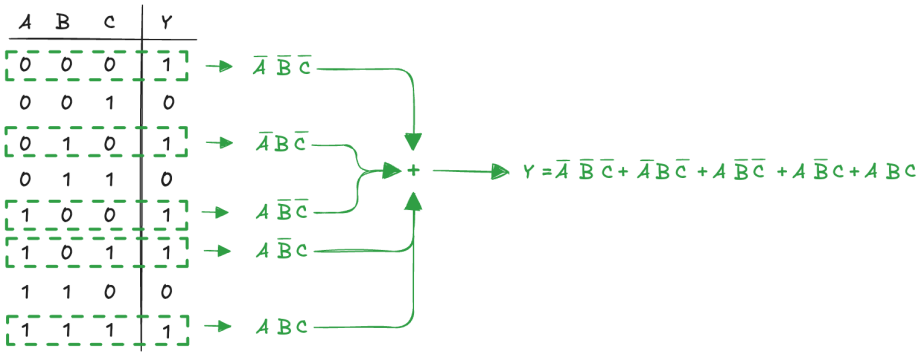


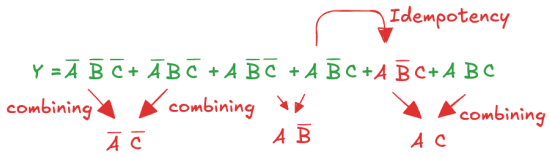
A floating node is **unpredictable**: it might be read as 0, as 1, or as an intermediate voltage. A common mistake that produces a floating node is to **leave an input pin unconnected**. In this case the circuit can **behave erratically**, switching randomly between values or even reacting to static charges from a person's touch. This explains why, in a lab environment, some circuits appear to work correctly only while a student's finger is resting on a chip, the touch slightly biases the floating input.

1.6 Exercises

1 - Write a Boolean equation in sum-of-products canonical form for the following truth tables of 3 variables, then simplify the equation using Boolean algebra rules

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





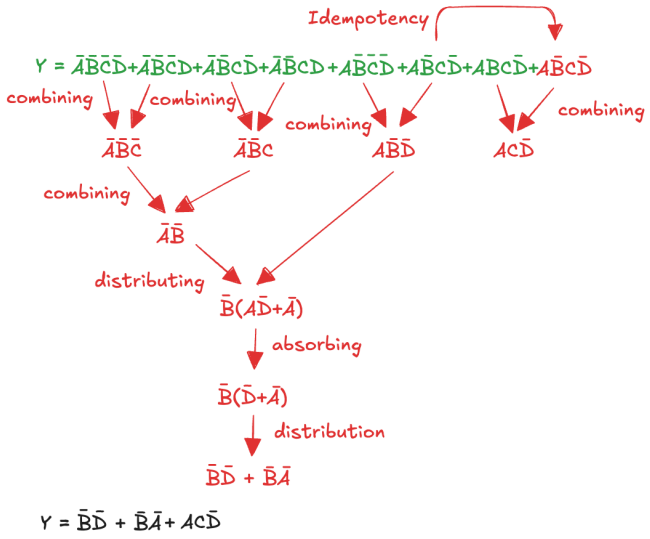
$$Y = \bar{A}\,\bar{C} + A\,\bar{B} + A\,C$$

2 - Write a Boolean equation in sum-of-products canonical form for the following truth tables of 4 variables, then simplify the equation using Boolean algebra rules

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

A	B	C	D	Y	
0	0	0	0	1	$\rightarrow \bar{A}\bar{B}\bar{C}\bar{D}$
0	0	0	1	1	$\rightarrow \bar{A}\bar{B}\bar{C}D$
0	0	1	0	1	$\rightarrow \bar{A}\bar{B}C\bar{D}$
0	0	1	1	1	$\rightarrow \bar{A}\bar{B}CD$
0	1	0	0	0	
0	1	0	1	0	
0	1	1	0	0	
0	1	1	1	0	
1	0	0	0	1	$\rightarrow A\bar{B}\bar{C}\bar{D}$
1	0	0	1	0	
1	0	1	0	1	$\rightarrow A\bar{B}C\bar{D}$
1	0	1	1	0	
1	1	0	0	0	
1	1	0	1	0	
1	1	1	0	1	$\rightarrow ABC\bar{D}$
1	1	1	1	0	

$$\rightarrow \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}C\bar{D} + \bar{A}\bar{B}CD + A\bar{B}\bar{C}\bar{D} + A\bar{B}C\bar{D} + ABC\bar{D}$$



$Y = \bar{B}\bar{D} + \bar{B}\bar{A} + AC\bar{D}$