Boolean Algebra

CISC 2210 — CUNY Brooklyn College
Lecture Notes

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Boolean Algebra

Boolean algebra is a branch of mathematics that deals with variables that have two distinct values: typically represented as **true** (1) and **false** (0).

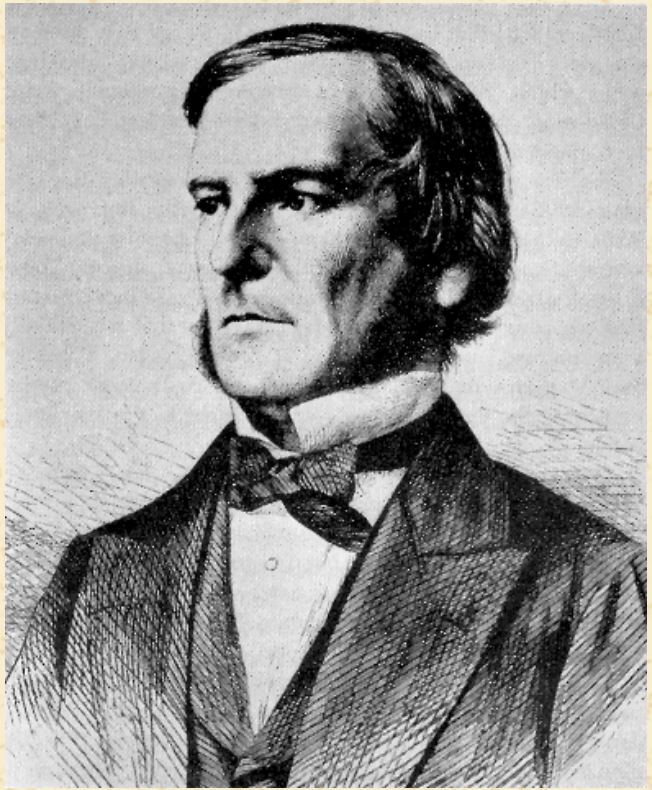
Developed by the English mathematician George Boole in the mid-19th century and described in his 1854 book, Boolean algebra provides a formal framework for analyzing and simplifying logical expressions and reasoning about logical operations. Boolean algebra is foundational in computer science, digital circuit design, and logic.

Boolean expressions are mathematical expressions that involve Boolean variables (= bits) and logical operations. They can represent logical propositions and can be evaluated to yield a Boolean value (true or false). A Boolean expression can be simple or complex, depending on the number of variables and operations involved.

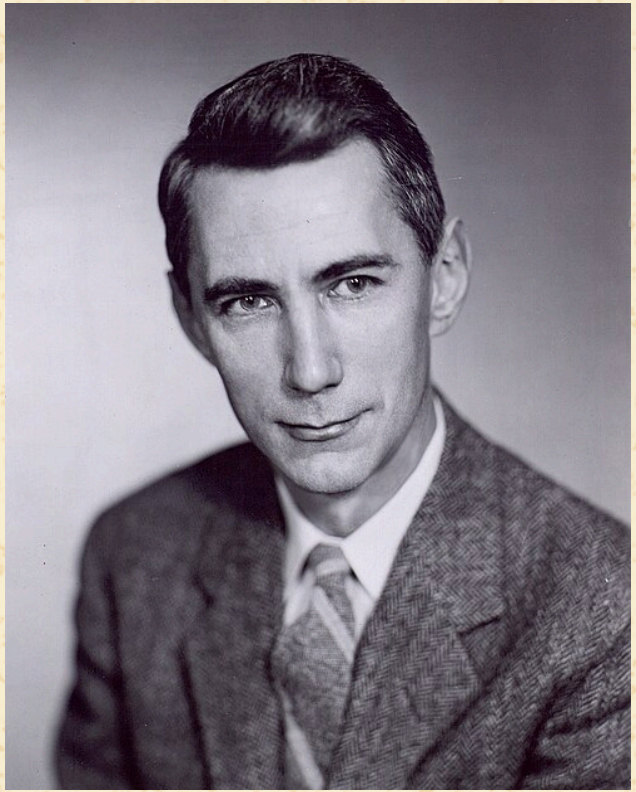
Later, in the 20th century, Boolean Algebra became fundamental in electrical engineering and digital circuit design. The American mathematician and electrical engineer Claude Shannon showed that Boolean expressions can model switching circuits, and this connected logic with hardware.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Boolean Algebra



George Boole. George Boole - Public domain portrait engraving: picryl.com, Public Domain.



Claude Elwood Shannon. Tekniska Museet of Sweden, Item 43069 via Flickr, CC BY 2.0, via Wikimedia Commons.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Boolean Algebra: Uses

Boolean Algebra is widely used in both mathematics and computer science, and it provides a formal way to reason about truth values:

- In mathematics, it appears in logic, set theory, and proof systems, and it supports reasoning about propositions.
- In computer science, it is used in programming, algorithms, and database queries. It also plays a central role in digital circuit design, where logical operations correspond to physical gates, as we shall see soon in these slides.

Boolean expressions are used in conditional statements such as `if` and `while`.

Thus, Boolean Algebra connects abstract reasoning with practical computation.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Rules of Boolean Algebra

A Boolean Algebra is a mathematical structure consisting of (1) a set together with (2) operations and distinguished elements:

- The set typically contains elements such as 0 and 1, which represent false and true.
- The main operations are conjunction $\land$ (`$\land$`), disjunction $\lor$ (`$\lor$`), and negation/inversion $\neg$ (`$\neg$`).
  - The operation $\land$ represents logical AND, and it returns true only when both operands are true.
  - The operation $\lor$ represents logical OR, and it returns true when at least one operand is true.
  - Lastly, the operation $\neg$ represents logical NOT, and it inverts the truth value of an element.

Example: $1 \land 0 = 0$, and $1 \lor 0 = 1$. Also, $\neg 1 = 0$, and $\neg 0 = 1$. Formally, a Boolean Algebra can be written as $(B, \lor, \land, \neg, 0, 1)$.

Interesting observation: we can substitute $\land$ with $\cdot$ (`$\cdot$`) (multiplication) and $\lor$ with $+$ (addition), and have these operations work *almost* completely fine as with integer arithmetics: $0 \cdot 0 = 0$, $0 \cdot 1 = 0$, $1 \cdot 0 = 0$, $1 \cdot 1 = 1$, $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, but $1 + 1 = 1$.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Rules of Boolean Algebra

Boolean Algebra satisfies several important algebraic properties that govern its operations. Let x, y, and z be two Boolean variables. We see that:

- Commutativity holds, so $x \wedge y = y \wedge x$, and $x \vee y = y \vee x$.
- Associativity holds, so $(x \wedge y) \wedge z = x \wedge (y \wedge z)$.
- Distributivity also holds, so $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$.
- There are identity elements, where $x \wedge 1 = x$, and $x \vee 0 = x$.
- Boolean Algebra also includes complement laws that relate elements with their negations: for any element x, we have $x \vee \neg x = 1$.
- Similarly, $x \wedge \neg x = 0$, and this expresses the law of contradiction.
- Idempotent laws hold, so $x \vee x = x$, and $x \wedge x = x$.
- Absorption laws also hold, such as $x \vee (x \wedge y) = x$.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Boolean Identities

The following identities directly follow the logical equivalences on slide 23 of Topic 3.

Name	Conjunction ($\wedge$) version	Disjunction ($\vee$) version
Identity Law	$x \wedge 1 = x$	$x \vee 0 = x$
Null Law	$x \wedge 0 = 0$	$x \vee 1 = 1$
Idempotent Law	$x \wedge x = x$	$x \vee x = x$
Complement Law	$x \wedge \neg x = 0$	$x \vee \neg x = 1$
Commutative Law	$x \wedge y = y \wedge x$	$x \vee y = y \vee x$
Associative Law	$(x \wedge y) \wedge z = x \wedge (y \wedge z)$	$(x \vee y) \vee z = x \vee (y \wedge z)$
Distributive Law	$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
Absorption Law	$x \wedge (x \vee y) = x$	$x \vee (x \wedge y) = x$
DeMorgan's Law	$\neg(x \wedge y) = \neg x \vee \neg y$	$\neg(x \vee y) = \neg x \wedge \neg y$
Double Negation Law	$\neg(\neg x) = \neg\neg x = x$	

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Boolean Identities

The identities below are the same as on the previous slide but use the operations: $'$, $\cdot$, and $+$ instead of: $\neg$, $\wedge$, and $\vee$:

Name	Conjunction ($\cdot$) version	Disjunction ($+$) version
Identity Law	$1x = x$	$0 + x = x$
Null Law	$0x = 0$	$1 + x = 1$
Idempotent Law	$xx = x$	$x + x = x$
Complement Law	$xx' = 0$	$x + x' = 1$
Commutative Law	$xy = yx$	$x + y = y + x$
Associative Law	$(xy)z = x(yz)$	$(x + y) + z = x + (y + z)$
Distributive Law	$x + yz = (x + y)(x + z)$	$x(y + z) = xy + xz$
Absorption Law	$x(x + y) = x$	$x + xy = x$
DeMorgan's Law	$(xy)' = x' + y'$	$(x + y)' = x'y'$
Double Negation Law	$(x')' = x'' = x$	

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Boolean Algebra vs. Sets

Boolean Algebra can also be interpreted in terms of sets, and this provides an intuitive perspective.

In this interpretation, the conjunction $\wedge$ corresponds to intersection $\cap$ (`$\cap$`), and the disjunction $\vee$ corresponds to union $\cup$ (`$\cup$`).

The negation $\neg x$ corresponds to the complement of a set A^c (`$A^c$`).

The element 0 corresponds to the empty set $\emptyset$ (`$\emptyset$`), and 1 corresponds to the universal set U (`$U$`).

Thus, Boolean Algebra can be seen as an abstraction of set operations: Boolean Algebra is essentially a set equipped with operations that satisfy certain rules: note the similarity between the Boolean operations described on the previous slides and the set laws/identities on Slide 56 of Topic 2. This abstraction allows the same structure to model logic, circuits, and sets.

Next, we will discover how Boolean Algebra is used to build computer circuits that let a computer make decisions.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Boolean Algebra vs. Sets

Any questions so far?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

The Transistor

A **transistor** is a tiny electronic device that acts as a switch or amplifier for electrical signals. It is made of semiconductor material, usually silicon, and has three main parts: the **collector**, **base**, and **emitter**. By controlling the flow of electrical current through these parts, a transistor can either allow or block the current, similar to a switch.

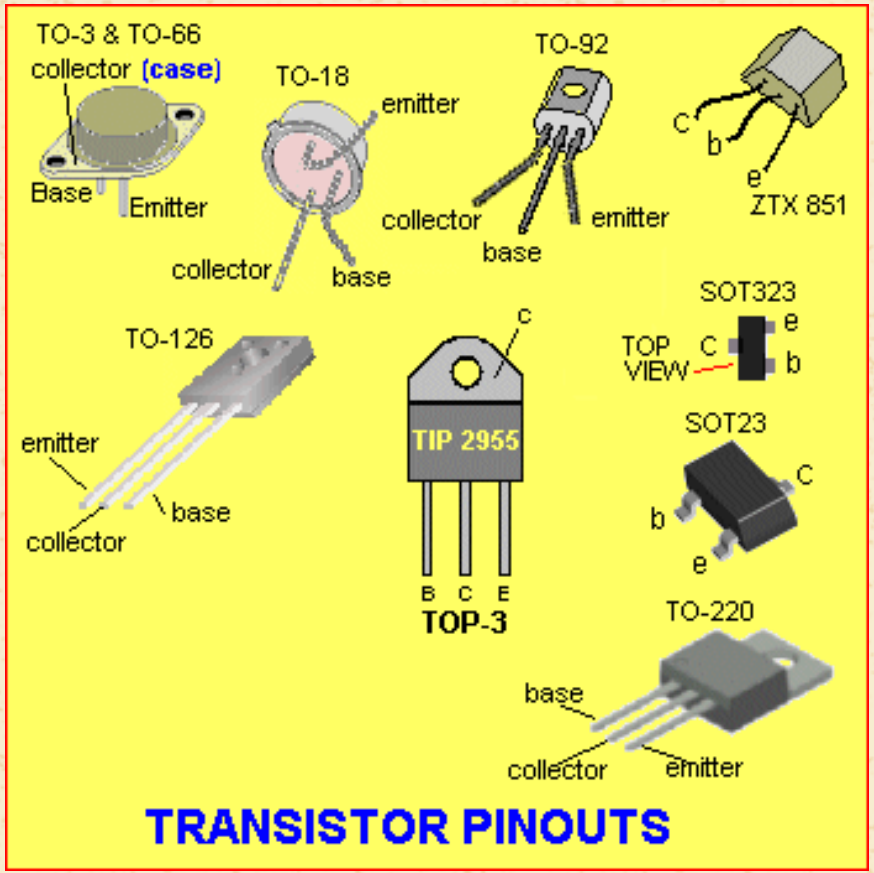
In computers, transistors are used to represent binary data (0s and 1s) by switching on (1) or off (0). This is the foundation of digital logic, which computers use to process information.

Transistors are incredibly small, so millions (even billions) can be packed into tiny chips like CPUs (central processing units). This has allowed computers to become faster, more powerful, and more energy-efficient. Transistors can be combined to create logic gates, memory units, and other components critical for computing functions.

Transistors can be combined to create logic gates, memory units, and other components critical for computing functions.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

The Transistor



Various transistors and their configurations. Taken from this article on talkingelectronics.com.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Logic Gates

Logic Gates are small and simple electronic circuits build up of transistors that make up the various hardware devices that a computer consists of: the CPU, RAM (= main memory), disks, etc., all consists of logic gates.

Generally, a logic gate takes either one or two inputs, each of which is a 0 or 1 bit, and output one bit (0 or 1). The output depends on (1) why kind of logic gate we use, and (2) what the input bits are.

Learning about logic gates is essential because they form the foundation of how computers and other digital systems process information, make decisions, and store data.

We will introduce the logic gates AND, OR, NOT, NAND, NOR, XOR, and XNOR.

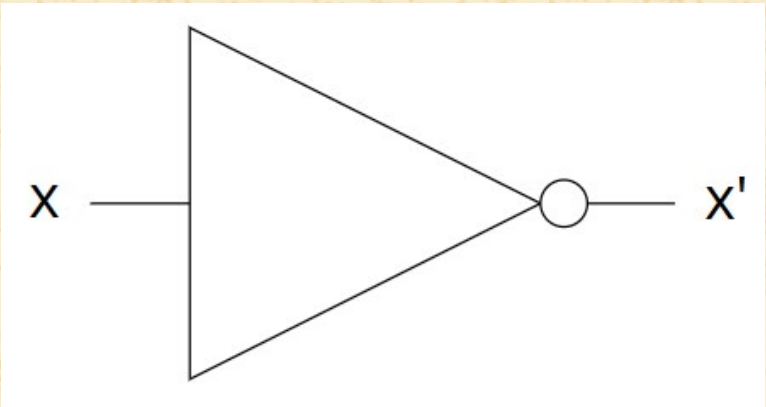
Many programming concepts, such as **conditional statements** (if-else) and **loops**, are rooted in the logic that is implemented by gates at the hardware level. Logic gates mirror the Boolean logic used in programming.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Logic Gates: NOT

The **NOT** logic gate, also called the **invertor** gate, takes one bit as an input, and outputs the opposite bit: for the input 0, it outputs 1, and for 1, it outputs 0. This gate acts exactly as the "not" operator in programming does: given the Boolean variable x, the symbol: $!x$ (`$!x$`), $\neg x$, or x' (`$x'$`) means "not x". You may also use the bitwise $\sim$ (`$\sim$`) operator: $\sim x$.

Below is the NOT gate diagram and truth table.



NOT Gate diagram. Miriam Briskman, CC BY-NC 4.0.

Input	Output
x	x'
0	1
1	0

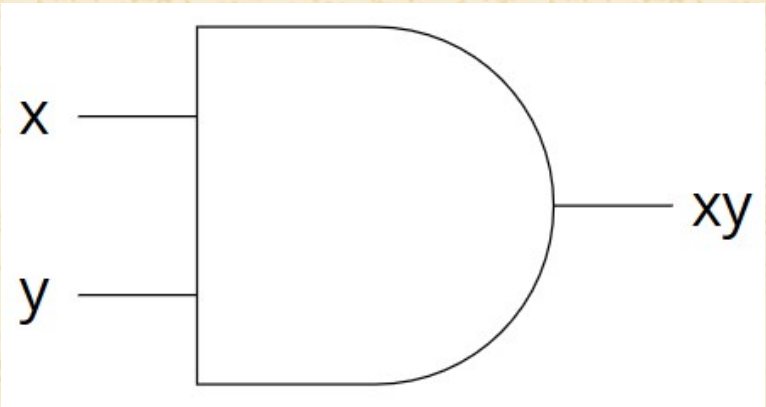
NOT truth table

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Logic Gates: AND

The **AND** logic gate takes two bits as an input. Its output follows the rule: "only if both bits are 1, output 1; otherwise, output 0." This gate acts exactly as the "and" operator in programming does: given the Boolean variables x and y, the symbol: $x \& \& y$ (`$x \&\& y$`), $x \wedge y$, $x \cdot y$, or xy means "x and y". You may also use the bitwise $\&$ operator: $x \& y$.

Below is the AND gate diagram and truth table.



AND Gate diagram. Miriam Briskman, CC BY-NC 4.0.

Input		Output
x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

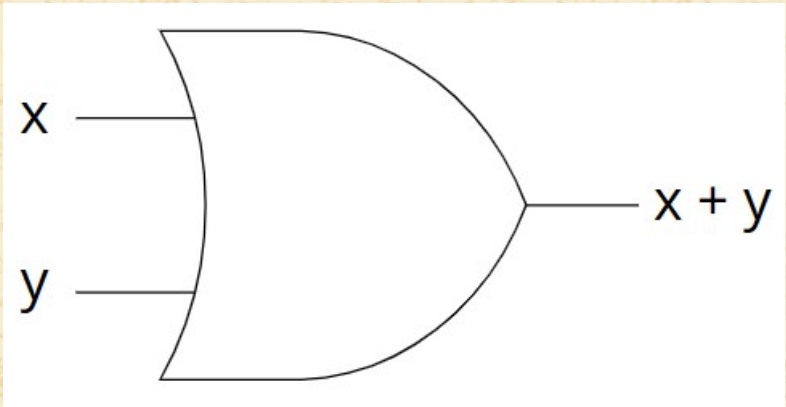
AND truth table

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Logic Gates: OR

The **OR** logic gate takes two bits as an input. Its output follows the rule: "if at least one of the bits is 1, output 1; otherwise, output 0." This gate acts exactly as the "or" operator in programming does: given the Boolean variables x and y, the symbol: $x||y$ (`$x || y$`), $x \vee y$, or $x + y$ means "x or y". You may also use the bitwise $|$ operator: $x|y$.

Below is the OR gate diagram and truth table.



OR Gate diagram. Miriam Briskman, CC BY-NC 4.0.

Input		Output
x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

OR truth table

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Logic Gates: OR

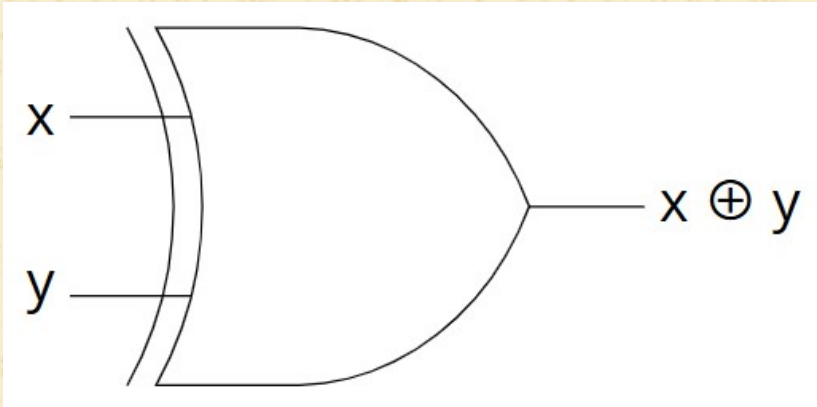
Any questions so far?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Logic Gates: XOR

The **XOR** (= "exclusive or") logic gate takes two bits as an input. Its output follows the rule: "if both bits are different, output 1; if they are the same, output 0." This gate acts exactly as the bitwise "xor" operator in programming does: given the Boolean variables x and y, the symbol: $x \wedge y$ (`$x ^\wedge y$`) or $x \oplus y$ (`$x \oplus y$`) means "x xor y".

Below is the XOR gate diagram and truth table.



XOR Gate diagram. Miriam Briskman, CC BY-NC 4.0.

Input		Output
x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

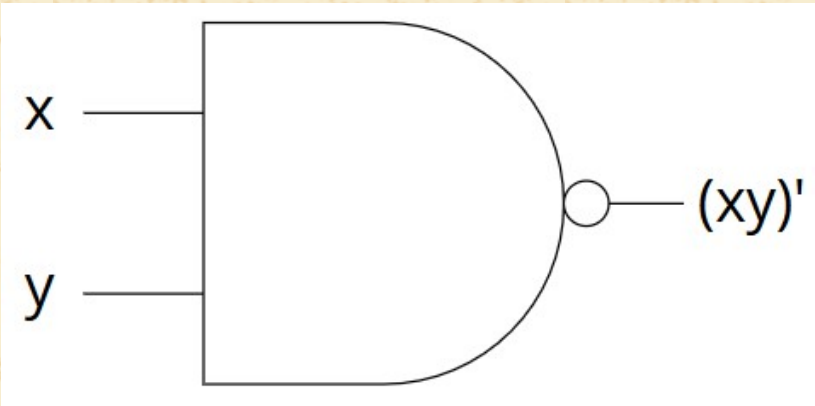
XOR truth table

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Logic Gates: NAND

The **NAND** (= "not and") logic gate takes two bits as an input. Its output follows the rule: "if the bits aren't both 1 (i.e., if at least one of bit is 0,) output 1; otherwise, output 0." You can implement it in your code as follows: given the Boolean variables x and y, the symbol: $!(x\&\&y)$ or $(xy)'$ means "x nand y". Below is the NAND gate diagram and truth table.

Bonus question: How can we build a NAND gate if we know how to build NOT, AND, OR, and XOR gates?



NAND Gate diagram. Miriam Briskman, CC BY-NC 4.0.

Input		Output
x	y	$(xy)'$
0	0	1
0	1	1
1	0	1
1	1	0

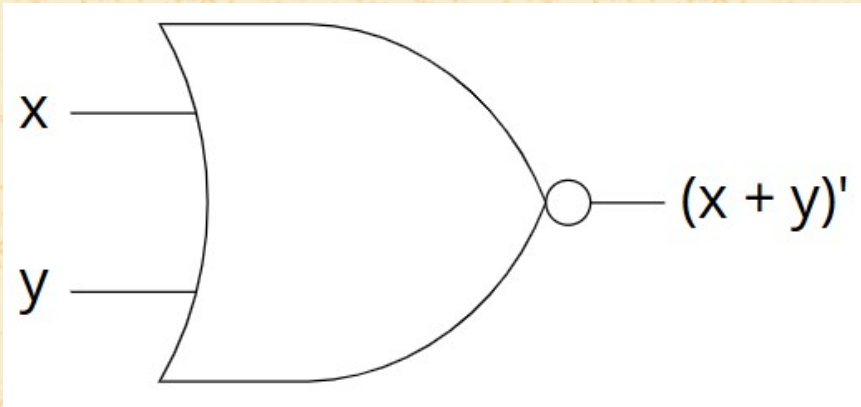
NAND truth table

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Logic Gates: NOR

The **NOR** (= "not or") logic gate takes two bits as an input. Its output follows the rule: "if both bits are 0, output 1; otherwise, output 0." You can implement it in your code as follows: given the Boolean variables x and y, the symbol: $!(x||y)$ or $(x+y)'$ means "x nor y". Below is the NOR gate diagram and truth table.

Bonus question: How can we build a NOR gate if we know how to build NOT, AND, OR, XOR, and NAND gates?



NOR Gate diagram. Miriam Briskman, CC BY-NC 4.0.

Input		Output
x	y	$(x+y)'$
0	0	1
0	1	0
1	0	0
1	1	0

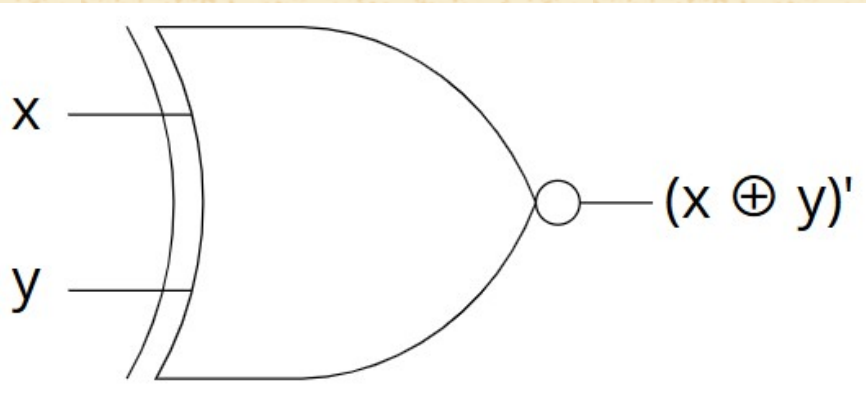
NOR truth table

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Logic Gates: XNOR

The **XNOR** (= "not exclusive or") logic gate takes two bits as an input. Its output follows the rule: "if both bits are the same, output 1; otherwise, output 0." You can implement it in your code as follows: given the Boolean variables x and y, the symbol: $!(x \hat{} y)$ or $(x \oplus y)'$ means "x xnor y". Below is the XNOR gate diagram and truth table.

Bonus question: How can we build an XOR gate if we know how to build NOT, AND, OR, XOR, NAND, and NOR gates?



XNOR Gate diagram. Miriam Briskman, CC BY-NC 4.0.

Input		Output
x	y	$(x \oplus y)'$
0	0	1
0	1	0
1	0	0
1	1	1

XNOR truth table

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Universal Gates: NAND and NOR

We can build any of the seven gates we introduced above using only NAND or only NOR gates, which makes each of these 2 gate types **universal**.

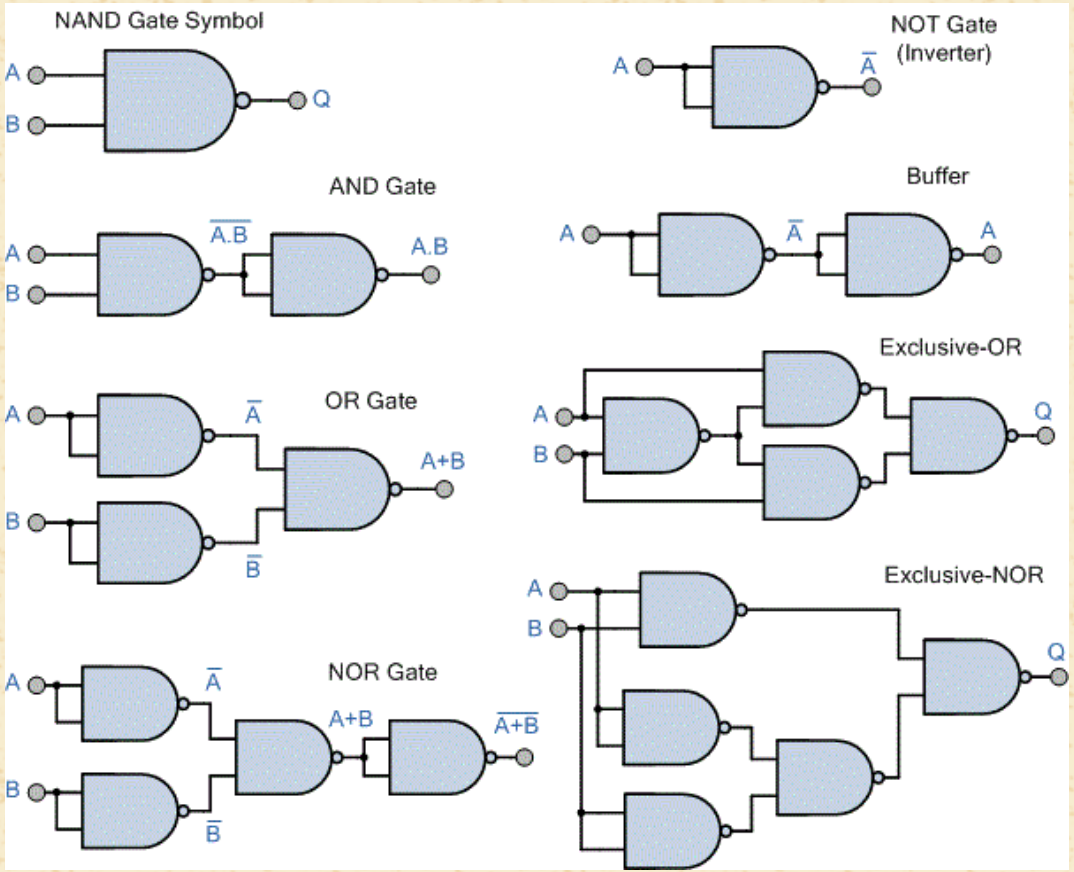
Computer circuit manufacturers benefit from this fact because:

1. It is cheaper to build NAND gates compared to other gates.
2. More complex circuits are easier to build with a single gate type vs. multiple gate types (NOT, AND, OR, etc.)

On the following slides, we prove that NAND and NOR are indeed universal gates by showing how we build each type of gate with them.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

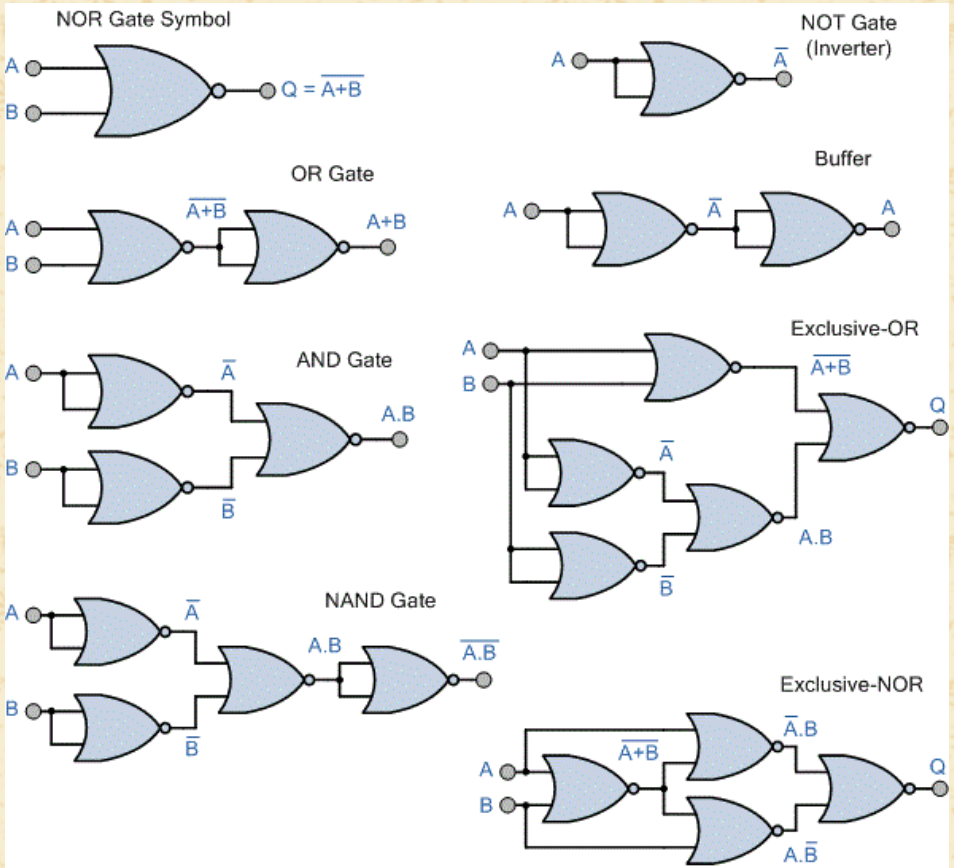
Universal Gates: NAND and NOR



Building all the gates from NAND. Taken from this Electronics Tutorial article on universal gates.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Universal Gates: NAND and NOR



Building all the gates from NOR. Taken from this Electronics Tutorial article on universal gates.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Universal Gates: NAND and NOR

Any questions so far?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Boolean Symbols and Expressions

Symbol summary:

Logic Gate	Boolean Expression
NOT x	x'
x AND y	$x \cdot y$ (or: xy)
x OR y	$x + y$
x XOR y	$x \oplus y$
x NAND y	$(xy)'$
x NOR y	$(x + y)'$
x XNOR y	$(x \oplus y)'$

We can combine these expressions to create more complex logic expressions, as we saw in Topic 3.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Boolean Symbols and Expressions

As we emphasized previously, a Boolean variable can be either "true" or "false". We can use Boolean variables to represent real-life events, such as "it will rain today" or "we will go watch the soccer game today". A complex logic expression will be one that combines one or more simple Boolean variables together using "not", "and", "or", etc., relations.

Example: The statement:

> It won't rain today, and we will go watch the soccer game today.

contains the two simple events "it will rain today" or "we will go watch the soccer game today". However, we first negated the 1st event (so it turned into "it won't rain today") and then ANDed it with the 2nd event.

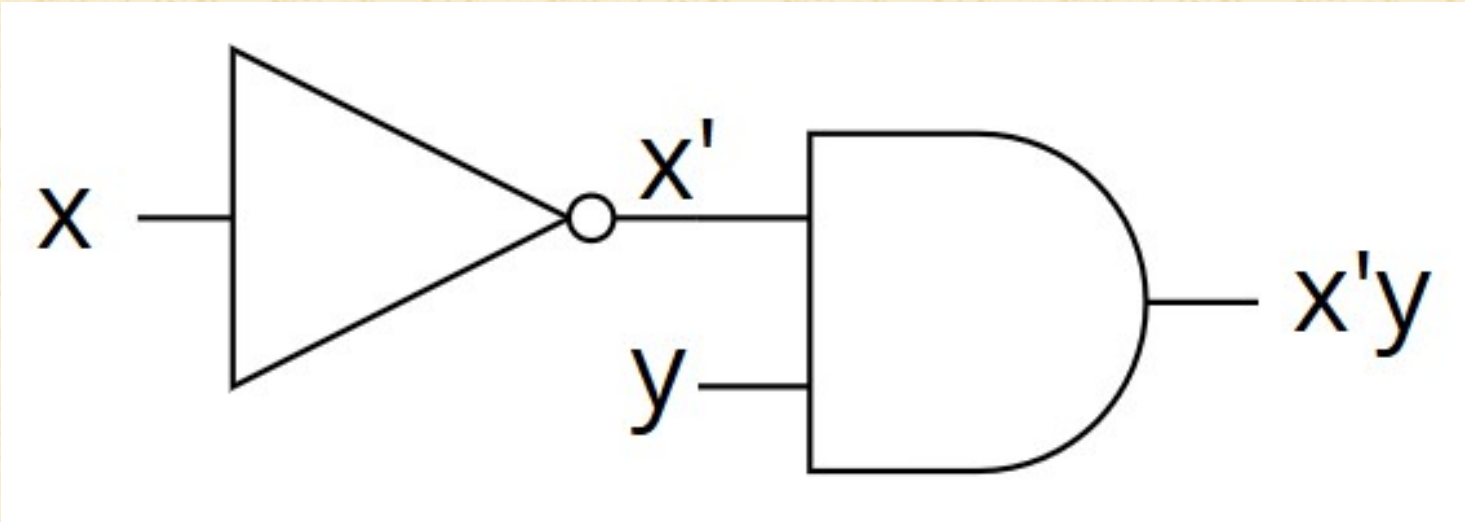
If we denote $x = (\text{"it will rain today"})$ and $y = (\text{"we will go watch the soccer game today"})$, then we can represent our statement with the expression: $x' \cdot y = x'y$.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Boolean Symbols and Expressions

We can turn a Boolean expression into a diagram of logic gates!

For example, the expression we got on the previous slide is $x' \cdot y = x'y = (x')y$ since we do the negation before doing the ANDing. We can take a NOT gate and feed the output from it into an AND gate, which results in the following diagram:



Building x'y using logic gates. Miriam Briskman, CC BY-NC 4.0.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Boolean Symbols and Expressions

Consider a different statement:

> If it doesn't rain today, then we will go watch the soccer game today.

This statement, which is written as "if x then y", can also be equivalently restated as "We will go watch the soccer game today if it doesn't rain today". (In this example, $x = (\text{"it won't rain today"})$ and $y = (\text{"we will go watch the soccer game today"})$.)

We don't have a logic gate for an "if x then y" situation, but notice that this statement is true either if x is false (regardless of what y is,) or if y is true (regardless of what x is.) In other words, the only case when this statement is false is when x is true and y is false (that is, when it doesn't rain, but, despite this, we don't go watch the game.)

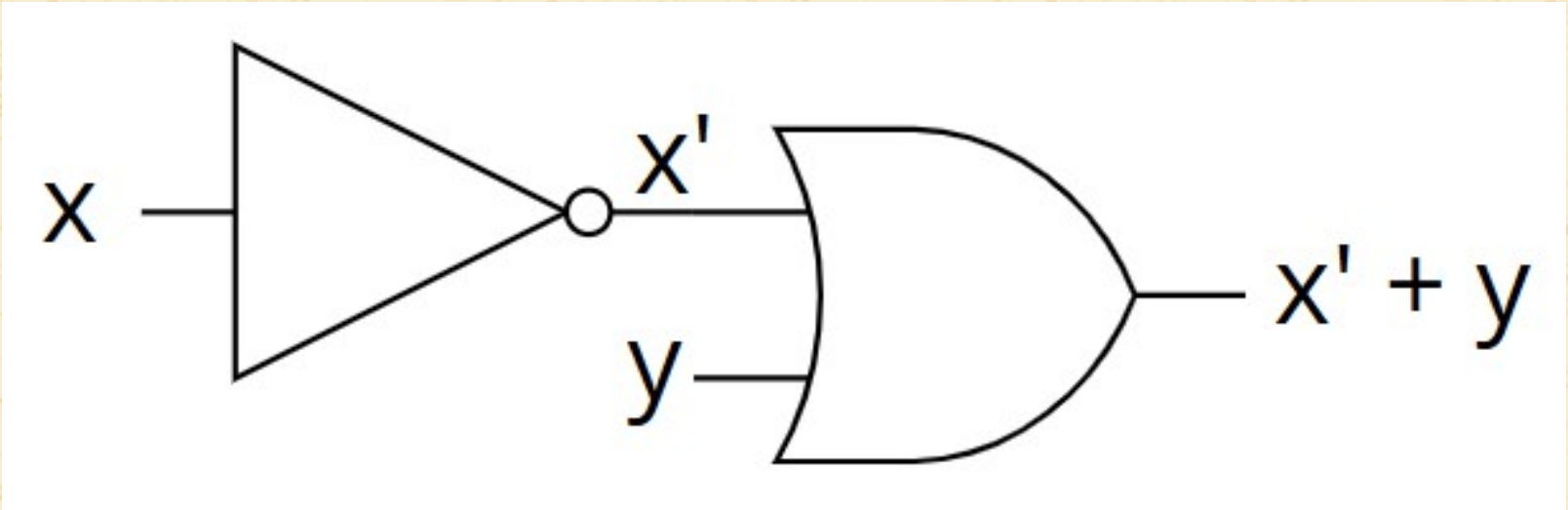
This description of the behavior of the "if x then y" statement tells us that we can express it as: $x' + y$ (not x, or y).

Conclusion: A statement of the form "if x then y" (or "y if x") can be expressed with the Boolean expression $x' + y$.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Boolean Symbols and Expressions

For the example on the previous slide, $x' + y$, we do the negation before doing the ORing. This leads to the following diagram:



Building x' + y using logic gates. Miriam Briskman, CC BY-NC 4.0.

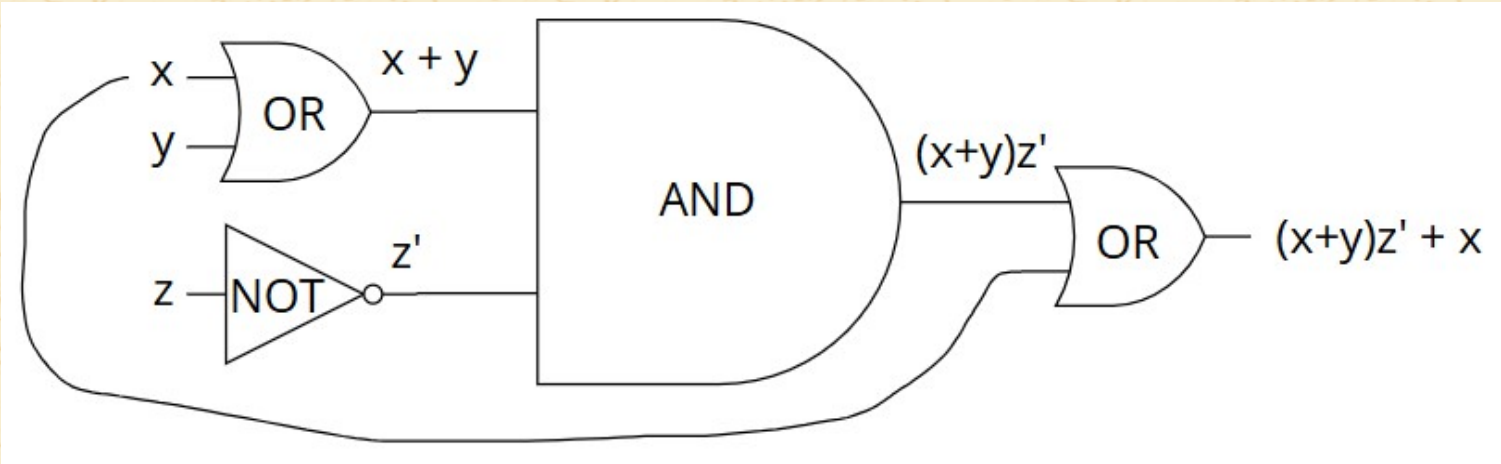
These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Boolean Symbols and Expressions

One more example: consider the statement:

> We can drink coffee or tea and not eat any donuts, or just drink coffee.

This statement sounds quite odd and abstract, but we can still express it with a Boolean expression. If we denote $x = (\text{"drink coffee"})$, $y = (\text{"drink tea"})$, and $z = (\text{"eat donuts"})$, the statement can be expressed as: $(x + y)z' + x$.



Building (x + y)z' + x using logic gates. Miriam Briskman, CC BY-NC 4.0.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Boolean Symbols and Expressions

Fun Class Exercise: Create logic gate diagrams for the following expressions:

1. $(x')'$ (***Bonus question***: what is this expression equal to?)
2. xyz
3. $x' + y'$
4. $(xy)'$
5. $(x \oplus y)z$

[Note about the precedence of operators: we will always do negation before ANDing; ANDing before XORing; and XORing before ORing unless we have parentheses, which we always do first.]

The diagrams in the previous slides were created using the following online logic gates diagram drawing app: https://online.visual-paradigm.com/app/diagrams/#diagram:type=LogicDiagram.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Boolean Symbols and Expressions

Any questions?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Boolean Algebra: Sources

Davis, Stephen. (2019). Chapter 7. *A Cool Brisk Walk Through Discrete Mathematics*, version 2.2. License: CC BY-SA: Attribution-ShareAlike.

Doerr, Al and Levasseur, Ken. (2024). Chapter 13. *Applied Discrete Structures*, https://discretemath.org/. License: CC BY-NC-SA: Attribution-NonCommercial-ShareAlike.

Sassolas, Mathieu. (2021). Chapter I.B. *Introduction to Discrete Mathematics: An OER for MA-471*, https://academicworks.cuny.edu/qb_oers/179/. License: CC BY-NC: Attribution-NonCommercial.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.