Boolean Algebra & Logic Gates

CISC 3310 — CUNY Brooklyn College
Lecture Notes

Boolean Algebra & Logic Gates

In this chapter, we will:

1. Learn what transistors are, what kinds of transistors exist, and why they are important to computer architecture.
2. Introduce logic gates and their function in computing.
3. Work with Boolean expressions, and illustrate them with logic gate diagrams.
4. Find out what Karnaugh maps are, and use them to simplify Boolean expressions.

The Transistor

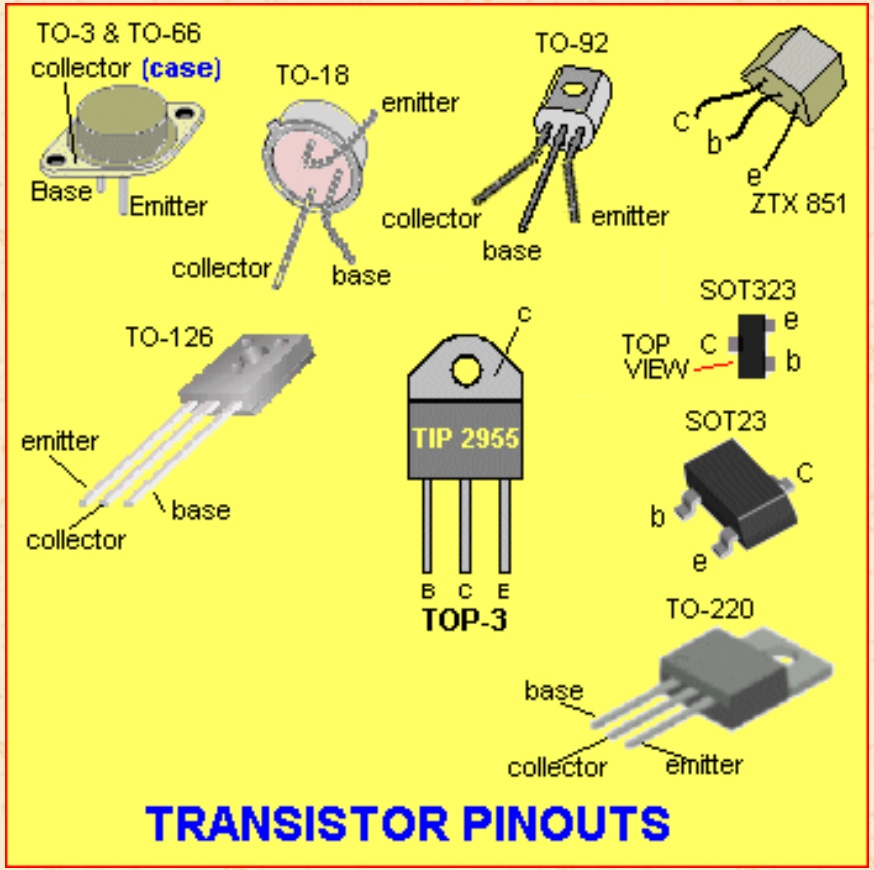
A **transistor** is a tiny electronic device that acts as a switch or amplifier for electrical signals. It is made of semiconductor material, usually silicon, and has three main parts: the **collector**, **base**, and **emitter**. By controlling the flow of electrical current through these parts, a transistor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

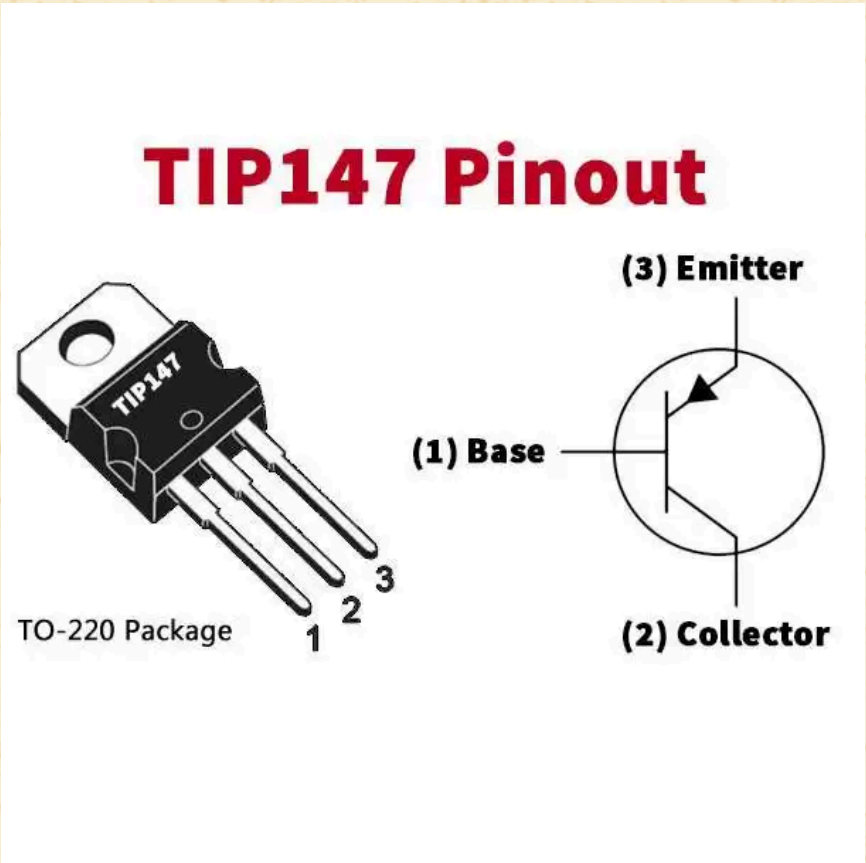
Transistors can be combined to create logic gates, memory units, and other components critical for computing functions.

The Transistor



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

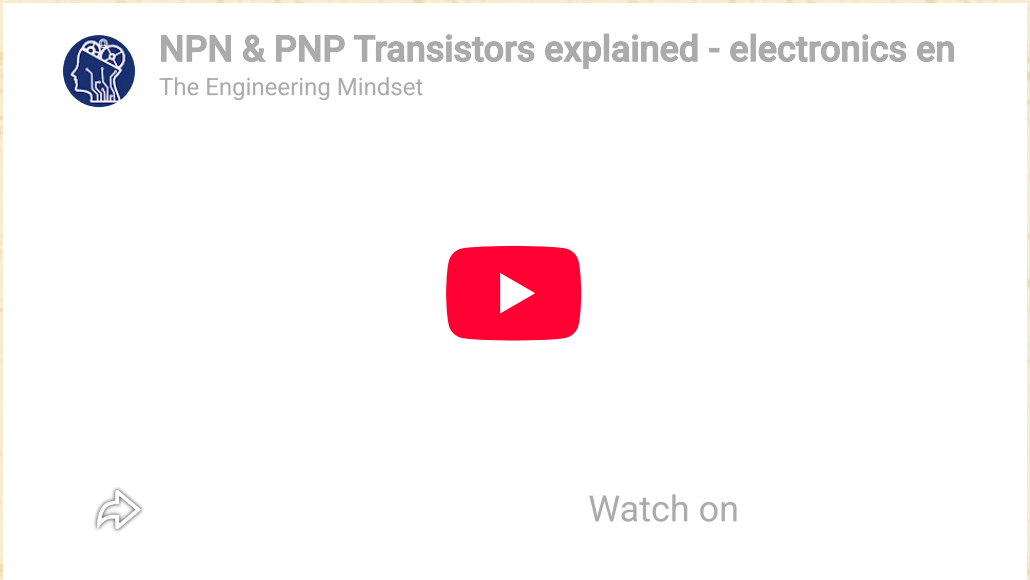
The Transistor



The TIP147 TO-220 Transistor. Taken from the makerselectronics.com online store.

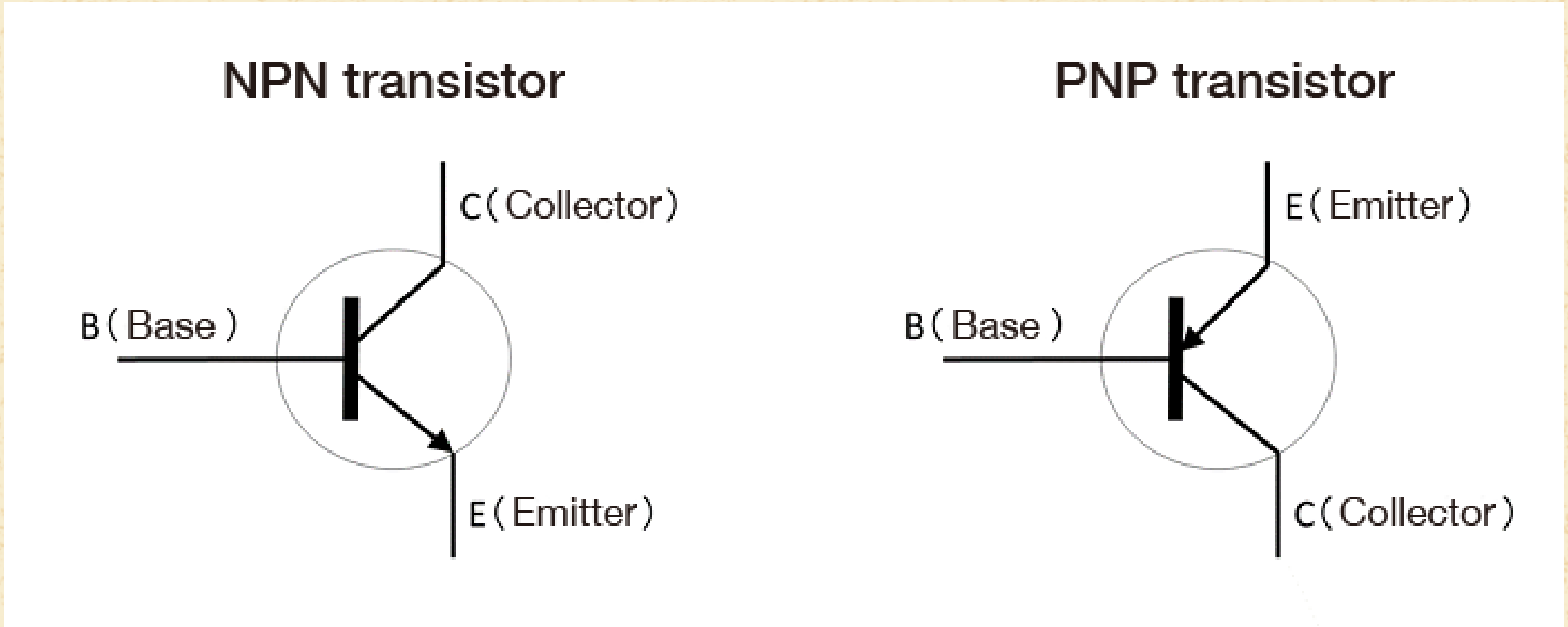
The Transistor

This video excellently explains how transistors work:



https://www.youtube.com/watch?v=zpyK5Hy8d0c

The Transistor



NPN vs. PNP Transistors. Taken from this article on www.hokuyo-aut.jp.

The Transistor

Transistors are primarily built from **semiconductor materials**, with **silicon** being the most common. Other materials used include **germanium** and, in specialized applications, **gallium arsenide** or **silicon carbide**. The choice of material affects how the transistor operates, particularly in terms of speed, efficiency, and power handling. **Silicon** is the most widely used material for transistors due to its excellent balance of performance, cost, and availability.

Transistors rely on the ability of semiconductors to **switch between conducting and non-conducting states**. Semiconductor materials like silicon can be precisely doped to control their electrical properties. In a transistor, the semiconductor's ability to either allow or block current (depending on applied voltage) is crucial for its switching and amplification functions.

By introducing impurities into the semiconductor material, manufacturers create **N-type** (electron-rich) and **P-type** (hole-rich) regions. These regions form **PN junctions**, which are critical for the operation of transistors. The controlled movement of electrons and holes across these junctions enables the transistor to either amplify or switch electrical signals.

The Transistor

Any questions so far?

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, 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.

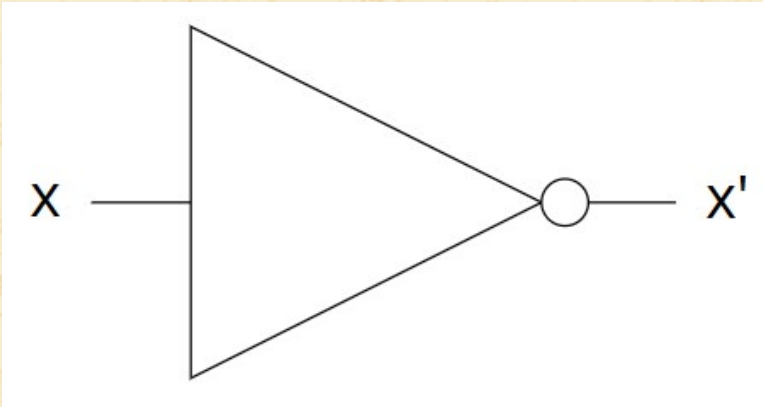
We will introduce the logic gates AND, OR, NOT, NAND, NOR, XOR, and XNOR.

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.

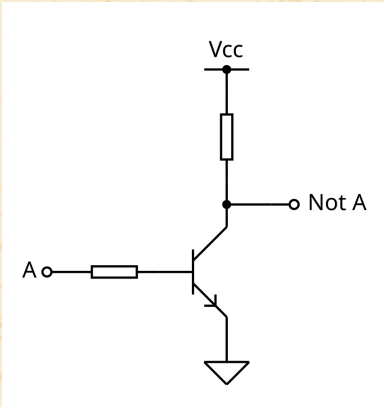
Logic Gates: NOT

The **NOT** logic gate, also known as the **invertor** gate, takes one bit as an input, and outputs the opposite bit: for the input 0, it outputs 1, and for the input of 1, it outputs 0. This gate acts exactly as the "not" operator in programming does: given the Boolean variable x, then: `!x` means "not x". You may also use the bitwise ~ operator: ~x.

Below is the graphical ("black-box") representation of the NOT gate, one way in which we can build it with transistors, and the truth table for this gate.



NOT Gate: black-box representation. Miriam Briskman, CC BY-NC 4.0.



One way to build a NOT Gate from transistors. Created by Tim Mathias, CC BY-SA 4.0, via Wikimedia Commons.

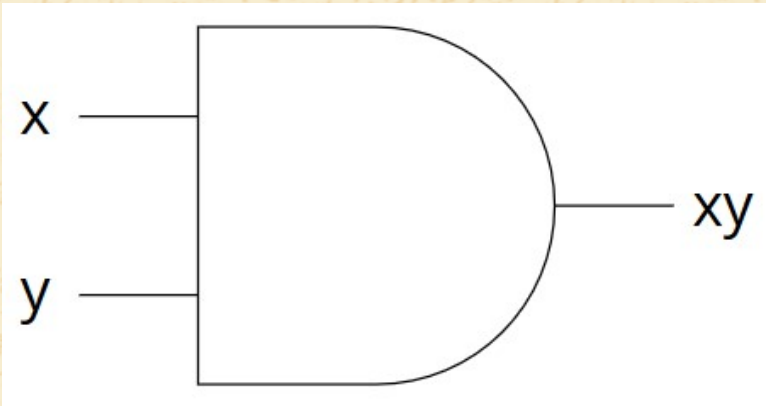
Input	Output
x	x '
0	1
1	0

NOT truth table

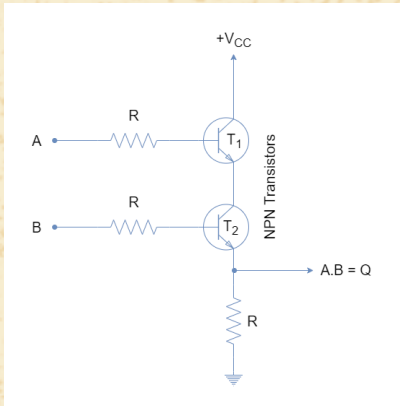
Logic Gates: AND

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, then `x && y` means "x and y". You may also use the bitwise & operator: `x & y`.

Below is the black-box representation of the AND gate, one way in which we can build it with transistors, and its truth table.



AND Gate: black-box representation. Miriam Briskman, CC BY-NC 4.0.



One way to build an AND Gate from transistors. Taken from this article by Muhammad Shahid.

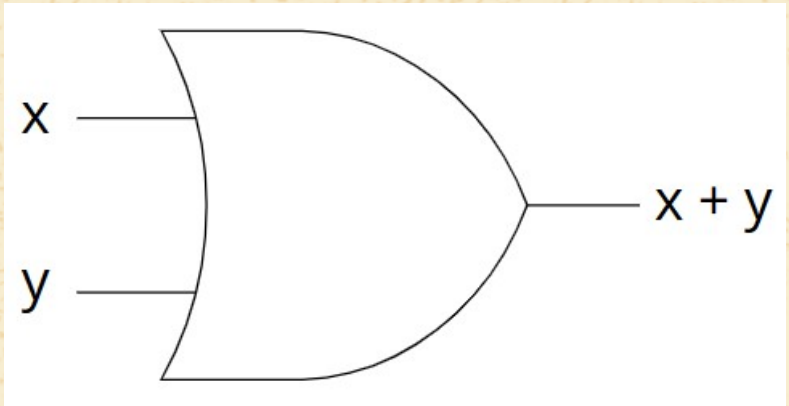
Input		Output
x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

AND truth table

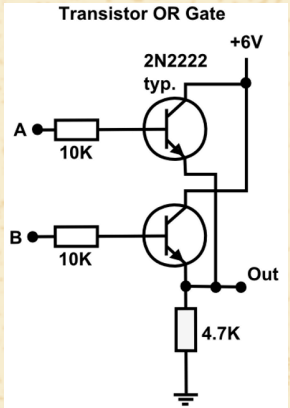
Logic Gates: OR

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, then: x || y means "x or y". You may also use the bitwise | operator: x | y.

Below is the black-box representation of the OR gate, one way in which we can build it with transistors, and its truth table.



OR Gate: black-box representation. Miriam Briskman, CC BY-NC 4.0.



One way to build an OR Gate from transistors. Taken from this article by nithya7rns.

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

OR truth table

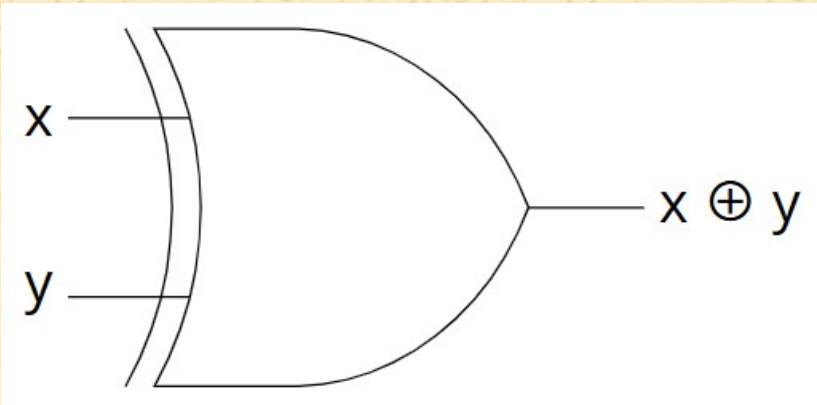
Logic Gates: OR

Any questions so far?

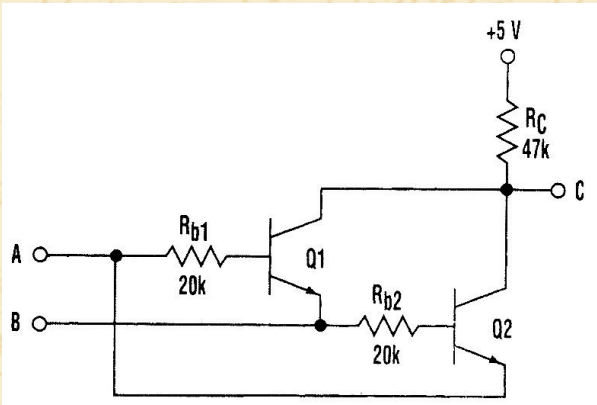
Logic Gates: XOR

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, then: x ^ y means "x xor y".

Below is the black-box representation of the XOR gate, one way in which we can build it with transistors, and its truth table.



XOR Gate: black-box representation. Miriam Briskman, CC BY-NC 4.0.



One way to build an XOR Gate from transistors. Taken from this article by Yann Guidon.

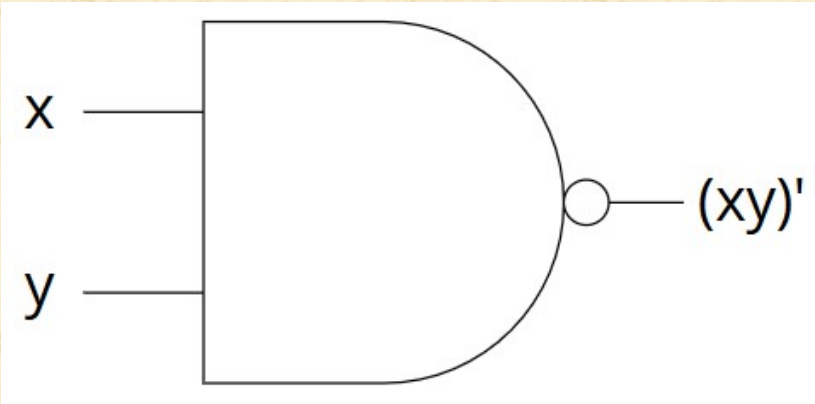
Input		Output
x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

XOR truth table

Logic Gates: NAND

The **NAND** (= "not and") logic gate takes two bits as an input. Its output follows the rule: "as long as the bits aren't both 1 (that is, if at least one of the bits is 0,) output 1; otherwise, output 0." You can implement it in your code as follows: given the Boolean variables x and y, then `!(x && y)` means "x nand y". Below is the black-box representation of the NAND gate and its 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: black-box representation. Miriam Briskman, CC BY-NC 4.0.

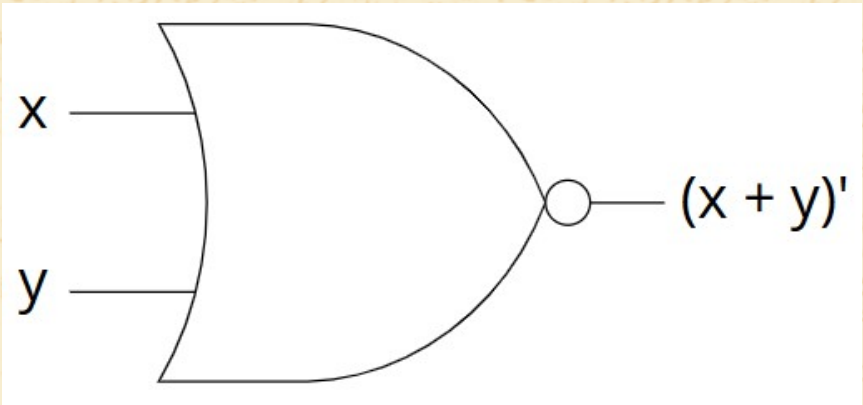
Input		Output
x	y	(xy)'
0	0	1
0	1	1
1	0	1
1	1	0

NAND truth table

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, then `!(x || y)` means "x nor y". Below is the black-box representation of the NOR gate and its 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: black-box representation. Miriam Briskman, CC BY-NC 4.0.

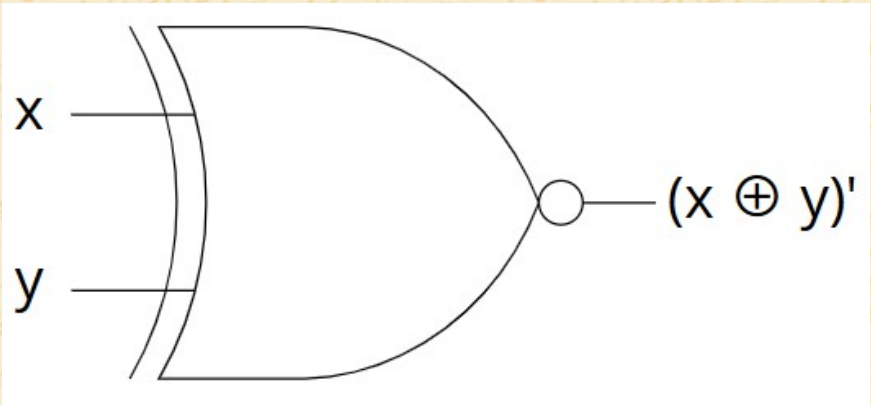
Input		Output
x	y	(x + y)'
0	0	1
0	1	0
1	0	0
1	1	0

NOR truth table

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, then `!(x ^ y)` means "x xnor y". Below is the black-box representation of the XNOR gate and its 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: black-box representation. Miriam Briskman, CC BY-NC 4.0.

Input		Output
x	y	(x ⊕ y)'
0	0	1
0	1	0
1	0	0
1	1	1

XNOR truth table

Universal Gates: NAND and NOR

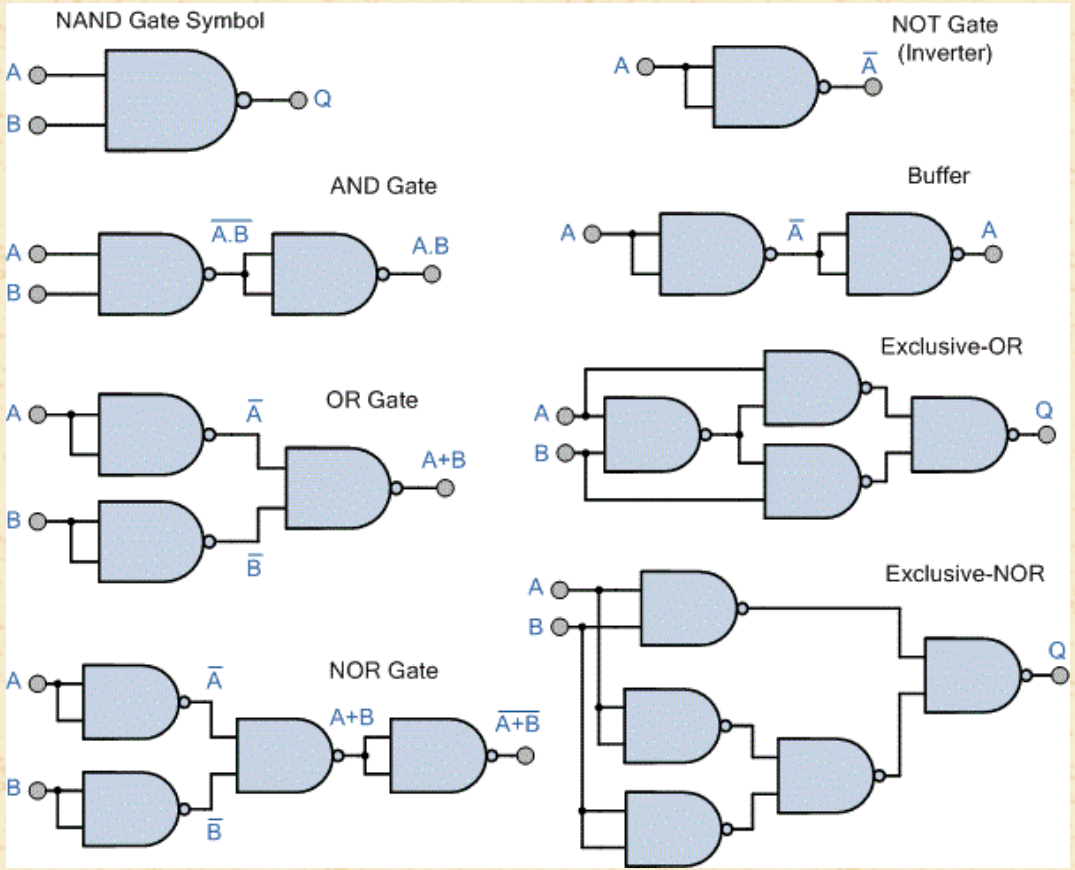
We can build any of the seven gates we introduced above using only NAND or only NOR gates, which makes each of these 2 gate types **universal**.

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.)

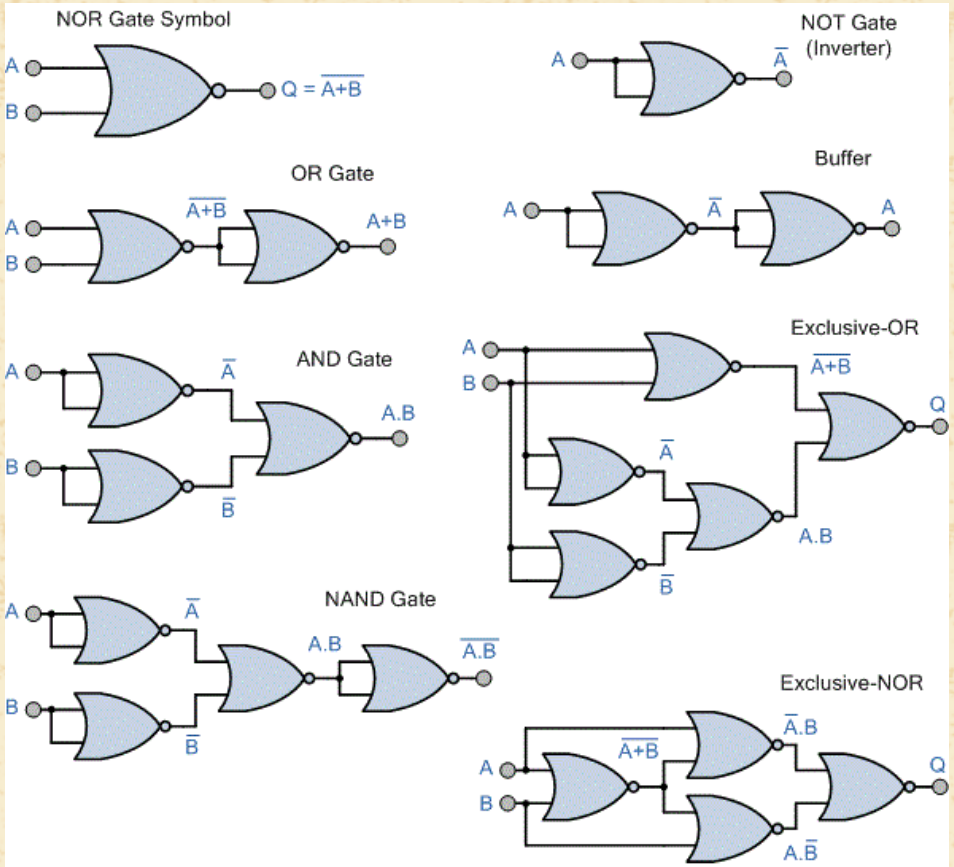
On the following slides, we prove that NAND and NOR are indeed universal gates by showing how we build each type of gate with them.

Universal Gates: NAND and NOR



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

Universal Gates: NAND and NOR



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

Universal Gates: NAND and NOR

Any questions so far?

Boolean Symbols and Expressions

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 mathematician George Boole in the mid-19th century, it 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.

On the next slide, we will list the Boolean expressions we use to represent the logic gates we introduced above.

Boolean Symbols and Expressions

We write:

Logic Gate	Boolean Expression
NOT x	x'
x AND y	x·y (or: xy)
x OR y	x + y
x XOR y	x ⊕ y
x NAND y	(xy)'
x NOR y	(x + y)'
x XNOR y	(x ⊕ y)'

We can combine these expressions to create more complex logic expressions.

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.

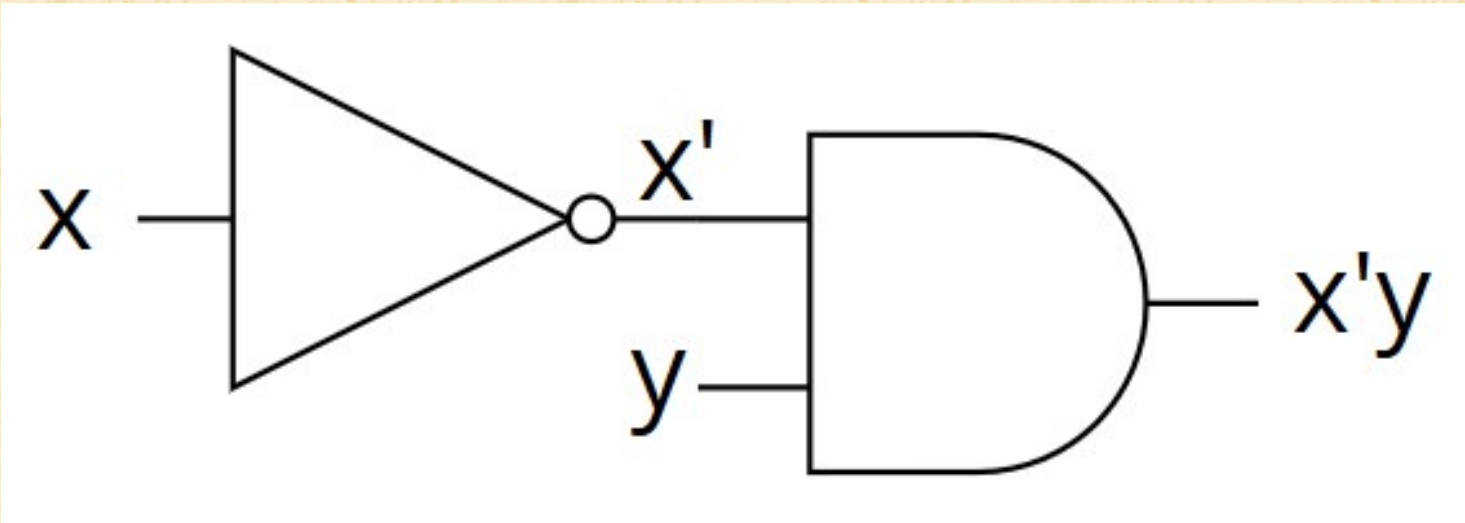
contains the two simple events "it will rain today" or "we will go watch the soccer game today". However, we first negated the 1st event (so it turned into "it won't rain today") and then ANDed it with the 2nd event.

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: $x' \cdot y = x'y$.

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.

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 $y = (\text{"we will go watch the soccer game today"})$.)

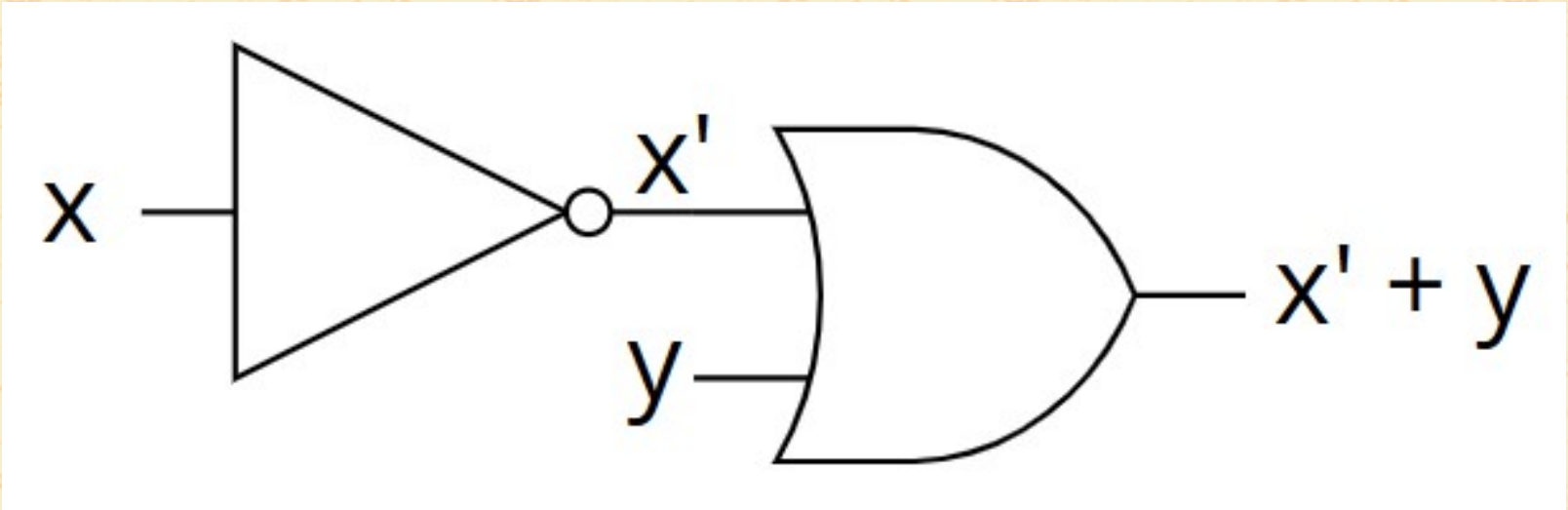
We don't have a logic gate for an "if x then y" situation, but notice that this statement is true either if x is false (regardless of what y is,) or if y is true (regardless of what x is.) In other words, the only case when this statement is false is when x is true and y is false (that is, when it doesn't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

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:



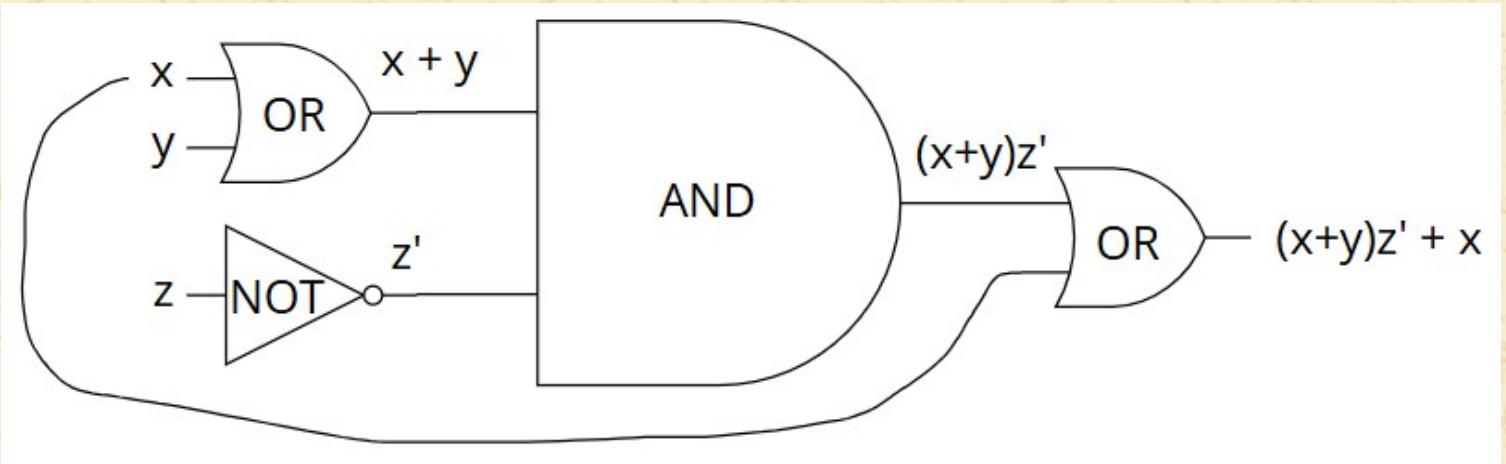
Building x' + y using logic gates. Miriam Briskman, CC BY-NC 4.0.

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.

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.

Boolean Symbols and Expressions

Any questions so far?

Boolean Identities

Sometimes, a given Boolean expression isn't written in its simplest form, but we can simplify it. This table contains several **identities**, which are pairs of expressions that are equivalent to each other and can be used to simplify Boolean expressions:

Name	AND version	OR version
Identity Law	$1x = x$	$0 + x = x$
Null Law	$0x = 0$	$1 + x = 1$
Idempotent Law	$xx = x$	$x + x = x$
Inverse 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 Complement Law	$(x')' = x'' = x$	

Boolean Identities

In the example below, we simplify the expression $(x + y)z' + x$ that we worked with on one of the previous slides:

$$\begin{aligned}(x + y)z' + x &= xz' + yz' + x \ [\text{Distributive Law: } (a + b)c = ac + bc] \\ &= yz' + xz' + x \ [\text{Commutative Law: } a + b = b + a] \\ &= yz' + x \ [\text{Absorption Law: } ab + a = a]\end{aligned}$$

This means that, instead of saying "We can drink coffee or tea and not eat any donuts, or just drink coffee," we could just say: "We can drink tea without eating any donuts, or just drink coffee," and still mean the same thing!

You can always use the following super-handy online expression simplifier whenever you want to know if a certain expression that you write in your `if`, `while`, etc., statement can be simplified (which will make your programs run faster!) https://www.boolean-algebra.com/.

Boolean Identities

Suppose you are given two expressions, e.g., $(x+y)'$ and $x'y'$, and want to check if they are identical or not (i.e., to verify if $(x+y)' = x'y'$ is a valid identity or not.) You can do so by constructing the **truth tables** for each of the 2 expressions!

x	y	$x+y$	$\boldsymbol{(x+y)'}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

x	y	x'	y'	$\boldsymbol{x'y'}$
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

Because the sequences of bits that we got under the $(x+y)'$ and $x'y'$ columns are exactly the same: $[1,0,0,0]$, this means that the expressions are equivalent, so $(x+y)' = x'y'$ is a valid identity!

Bonus point: What is the name of this identity?

Boolean Identities

Some rules (and tips) on building truth tables for Boolean expressions:

- The few leftmost columns of your table should be the simple variables the expression contains. For example, if your expression has the variables x, y, and z in it, your table should start with 3 columns, one for each of x, y, and z.
- The number of rows that your table will have can be calculated as follows: $2^{(\text{amount of variables})}$. For example, if your expression has the variables x, y, and z in it, the truth table will feature $2^3 = 8$ rows of bits.
- To make the evaluation of expressions gradual and easy, include additional columns in your table that correspond to intermediate expressions that build up the final expression. For example, if your expression is $(x + y)'$, don't include just the columns x, y, and then right away $(x + y)'$. Instead, include also a column for the OR operation $x + y$, which is an intermediate operation that you perform before applying the negation operation.
- The 'variables' 1 and 0 are special: while variables like x and y vary (each of them can be either 0 or 1,) 1 and 0 are actually constants. If you need to include a column for 1, all the bits across this column will be 1s. Similarly, if you need to include a column in the truth table for 0, all the bits across this column will be 0s.

Boolean Identities

- If you need to check whether two expressions are equivalent, you'll need to build 2 tables, one for each expression. You may, if wanted, 'glue' these tables to one another to create a single, longer table.
- Two expressions are equivalent if and only if the sequences of bits underneath the columns for each of these expressions are identical. In the example for $(x+y)'$ and $x'y'$, we saw that the sequence of bits for each of these was $1, 0, 0, 0$.
- If the sequences of bits differ from one another, even in just one of the bits, the two expressions are not equivalent.

Let's verify a couple more identities. The truth table for the identity $1x = x$ is:

1	$\boldsymbol{x}$	$\boldsymbol{1 \cdot x}$
1	0	0
1	1	1

Since the bits under x and under $1 \cdot x$ are the same (i.e., $[0, 1]$), we successfully confirmed the identity.

Boolean Identities

The truth tables for the identity $x + yz = (x + y)(x + z)$ are:

x	y	z	yz	$\boldsymbol{x + yz}$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

x	y	z	$x + y$	$x + z$	$\boldsymbol{(x + y)(x + z)}$
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	1	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

Because the sequences of bits that we got under the $x + yz$ and $(x + y)(x + z)$ columns are exactly the same: $[0, 0, 0, 1, 1, 1, 1, 1]$, this means that the expressions are equivalent, so $x + yz = (x + y)(x + z)$ is indeed a valid identity!

Boolean Identities

Let's check if the expressions: $x + yz$ and $y + xz$ are equivalent or on:

x	y	z	yz	$\boldsymbol{x + yz}$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

x	y	z	xz	$\boldsymbol{y + xz}$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	0	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	1
1	1	1	1	1

Because the sequence of bits that we got under the $x + yz$ column is different from the one under the $y + xz$ column, these two expressions aren't equivalent. We say: $x + yz \neq y + xz$.

Boolean Identities

Fun in-class exercises:

1. Confirm the following identities by building truth tables for them:
   a. $x(y+z) = xy + xz$
   b. $x + xy = x$
2. Check if the following pairs of expressions are equivalent or not by building truth tables for them:
   a. $(x+y)'$ and $x' + y'$
   b. $x'x + y'$ and $xy' + y'$

Note: for homework assignments and exams, if you are asked to create a truth table, you should use any computer app that can create tables, e.g., Microsoft Word, and then take a screenshot of it and attach to your hw/exam responses. In other words, don't submit sketches of truth tables that you draw by hand.

Boolean Identities

Any questions so far?

Boolean Identities: Computing the Runtime Savings

Suppose that a `while` loop in the source code of a program you are writing has the following header:

```
while ((x && y && z) || (x && y && !z) || (!x && y && z))
```

but you realized that the above condition is equivalent to simply saying:

```
while (y && (x || z))
```

How fast will the 2nd version of this code line run, in terms of the number of operations that we perform, compared to the 1st version?

The 1st conditional performs 2 negations, 6 ANDings, and 2 ORings, which are 10 operations in total. The 2nd conditional performs 1 ORing and 1 ANDing only. So, if all these operation types take the same time to run, the 2nd version is $\frac{10}{2} = 5$ times faster!

Boolean Identities: Computing the Runtime Savings

Any questions so far?

Expression Form Types

We've seen that expressions may have several equivalent forms.

To prevent confusion, computer scientists prefer to stick to specific types of forms which they call **standardized** or **canonical** forms. These types include:

- The **sum-of-products** form, e.g. $E = xy + x'y'$.
  - It's an expression of ANDed variables that are then ORed together.
- The **product-of-sums** form, e.g. $E = (x + y')(x' + y)$.
  - It's an expression of ORed variables that are then ANDed together.

The following is true and very useful to know:

- Every expression can be written in a sum-of-products form a unique product-of-sums form.
- The sum-of-products form is easier and preferred over the product-of-sums form.
- One can easily use an expression's truth table to write its sum-of-products form.

Expression Form Types

Example: Here is the truth table for some expression E:

x	y	z	E
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

To write the expression in a sum-of-products form, we look at all the cases when the expression E is **true** (= equal to 1,) which happen when: (1) $x = 0, y = 1$, and $z = 0$; (2) $x = 0, y = 1$, and $z = 1$; and (3) $x = 1, y = 1$, and $z = 1$.

Expression Form Types

This means that the expression E is true when $x'yz'$, or when $x'yz$, or when xyz.

We can combine these 3 instances using the OR (+) symbol: $E = x'yz' + x'yz + xyz$.

Note that the sum-of-products form isn't the simplest (= shortest) form of an expression: we can further simplify $x'yz' + x'yz + xyz$ using the identities to get a simpler sum-of-products form: $x'y + yz$.

On homework and exams, when asked to find the sum-of-products form from a truth table, you don't need to simplify the expression you got. If you are asked to simplify it further, you should use the tool at https://www.boolean-algebra.com/.

Another way to simplify Boolean expressions is by using handy tables called **Karnaugh maps**, which will be the last topic of these lecture notes.

Expression Form Types

Fun in-class exercise: Write the sum-of-products expression for the expression E represented by the truth table below:

x	y	z	E
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Reminder: To build the expression, look at the instances when E is **true** (= equal to 1.)

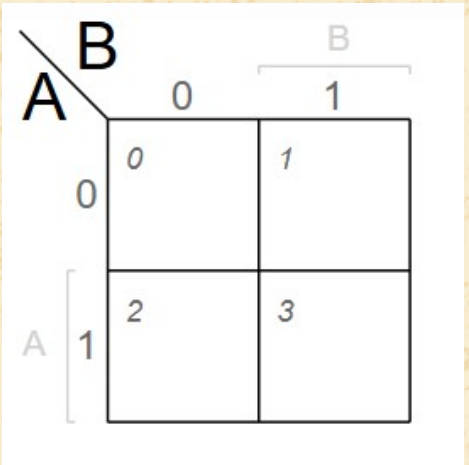
Expression Form Types

Any questions so far?

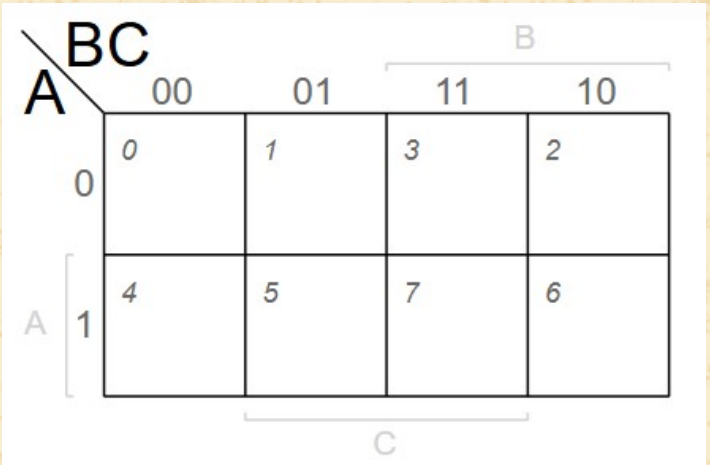
Kmaps

A **Karnaugh map** or **Kmap** is a table, in which each cell contains a bit and represents a single row from an expression's truth table. Kmaps are an easy tool for simplifying expressions.

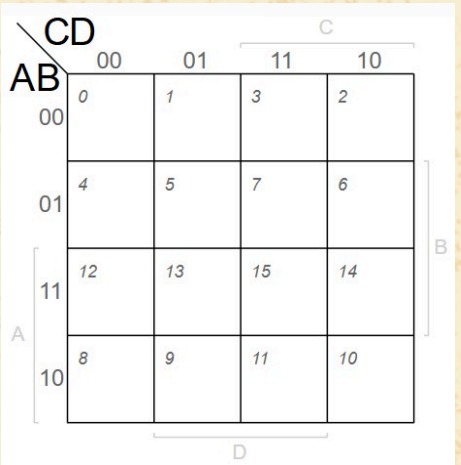
Here is how empty Kmaps for 2, 3, and 4-variable expressions look like:



A 2-variable expression Kmap. Based on this website.



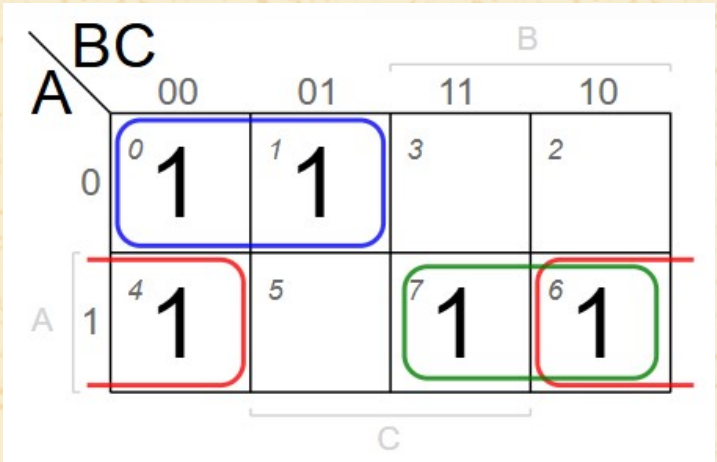
A 3-variable expression Kmap. Based on this website.



A 4-variable expression Kmap. Based on this website.

Kmaps

Here is an example of a Kmap for a 3-variable expression with bits filled in:



A 3-variable expression Kmap. Based on this website.

In this table, every 'empty' cell is treated as though it contains a 0. Ignore the colorful lines at this point.

Kmaps

To use a Kmaps to simplify an expression described by a truth table, we need to complete the following 4 steps:

1. Notice how many variables the expression/truth table contains, and select the correct Kmap based on this number.
2. Fill the cells of the Kmap with 1s and 0s according to column E in the truth table.
3. Encircle groups of adjacent cells of 1s, and
4. Write down the expression corresponding to those groups of 1s.

Let's explain how to complete each of these steps:

1. Given a truth table, check how many variables are involved. The number of variables is equal to the number of columns in the truth table minus 1 (since we don't count the column for E as a variable.) Then, pick up the right Kmap to use (either a 2-variable, 3-variable, or 4-variable Kmap.)
    - For example, if the truth table describes an expression using x and y, use a 2-variable Kmap.

Kmaps

2. Read the truth table, and, in each instance when the expression E is true, put 1 inside the corresponding cell in the Kmap.

   Note that the order of the cells in the Kmap doesn't exactly correspond to the order of rows in the truth table.

   For example, in the case of a 3-variable expression, the values of E from the top 4 rows of the truth table should be placed in the Kmap in the following order:

Bit from row 0	Bit from row 1	Bit from row 3	Bit from row 2

   That is, instead of adding the bits from rows 0, 1, 2, and then 3, the 1st row of the Kmap will contain the bits from rows 0, 1, 3, and then 2.

   Tip: The tiny numbers inside the Kmap diagrams' cells tell you what rows each cell corresponds to!

Kmaps

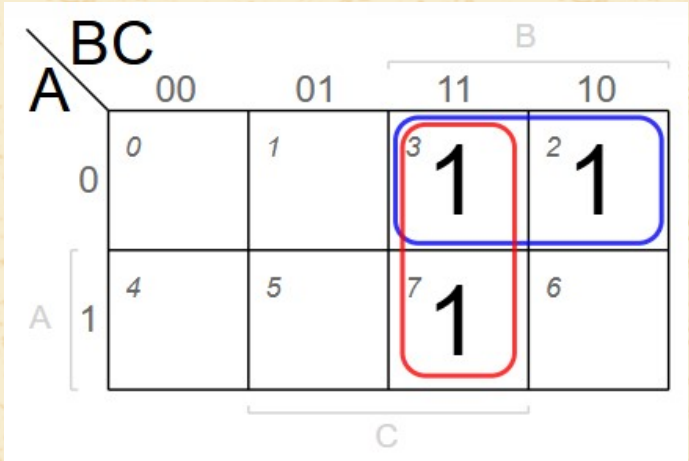
Example so far: Here is the truth table for the expression from slide 44:

x	y	z	E
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

The table contains 4 columns (and describes an expression that uses 3 types of variables, x, y, and z,) so we'll create a 3-variable Kmap.

Kmaps

The previous truth table, as we already know, shows that there are 3 cases in which E is true. This means that our Kmap will contain exactly three 1s (the rest will be 0s/empty cells.)



The Kmap for our example. Based on this website.

Tip: Note that the truth table rows with 1s are $\#2$, $\#3$, and $\#7$, so we put our 1s in the cells labeled with these exact numbers.

Kmaps

Note: The online tool at https://madformath.com/calculators/digital-systems/karnaugh-maps/karnaugh-map-solver, which we use in our examples, calls the Boolean variables A, B, C, D, etc., while we call our Boolean variables x, y, z, etc.

This naming doesn't affect the validity of the Kmap: you can name your Boolean variables whatever you wish. However, if a homework or exam question gives you a truth table that uses variables x, y, z, etc., you should write an answer that uses these exact variables and not A, B, C, D, etc.

That is, if a question asks you to use the above online tool, you will need to re-name your variables from x, y, z into A, B, C in order to use the Kmap, and then rename the variable in the final result from A, B, C back to x, y, z.

Kmaps

3. Once you fill up the Kmap with the correct bits, it's time to encircle groups of 1s according to the following principles:
    1. The groups must be of cells that contain only 1s.
    2. The number of cells in the group must be a power of 2, both vertically and horizontally. For example, valid groups are of the dimensions 1×1, 1×2, 2×1, 2×2, 1×4, 4×1, 4×2, etc. A group of the size 1×3, for example, isn't valid since 3 is not a power of 2.
    3. The groups must be built up of rows and columns: they can't be built through diagonals.
    4. The groups must be made as large as possible. For example, a row in a Kmap that contains four 1s must be grouped as 4×1, not as two separate groups of 2×1.
    5. Each of the 1s in the Kmap must belong to at least one group that you create: don't leave any 1 without a group.
    6. Groups can overlap with one another. Also, groups can 'wrap-around' the opposite sides of the Kmap.

Back to our example and Kmap on slide 53, we created two groups: the blue 1×2 group and the red 2×1 group.

Kmaps

4. It's time to build up the expression using the groups of 1 that we got.

   Each one of the groups you created will correspond to one term/product in the sum-of-products formula that we built up. For example, if you created 3 groups in a certain Kmap, blue, red, and green, then the sum-of-products formula will look like:
   $E =$ blue $+$ red $+$ green.

   Back to our example, we notice that the blue group corresponds to the case when B is true (= 1) and A is false (= 0) [and C is both 0 and 1, so we disregard C in this term], and that the red group corresponds to the case when both B and C are 1s [and A is both 0 and 1, so we disregard A in this one term].

   Hence, our expression becomes $A'B + BC$, which, when converted to the x, y, and z variables, is $x'y + yz$, which is the exact one simplified expression that we listed at the middle of slide 45!

Kmaps

Note: Since our assignments and exams are open-book, you won't be asked to encircle the groups of 1 yourself: you will be asked to use a specific tool, like the one at https://madformath.com/calculators/digital-systems/karnaugh-maps/karnaugh-map-solver to complete steps 3 and 4 of the method we described on slide 50.

You will, however, need to:

1. Decide which Kmap to use (based on the number of variables.)
2. Populate the Kmap according to the given truth table.
3. Enter the info of that Kmap into the online tool, which will encircle the groups of 1 and output the sum-of-products expression for you.
4. Rename the variables in the expression from A, B, C back to x, y, and z.

Kmaps

Fun in-class exercise: Use this online tool to write the simplified the sum-of-products expression for the expression E represented by the truth table below:

x	y	z	E
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Hint: Perform the steps listed on slide 57.

Kmaps

Any questions so far?

Boolean Algebra & Logic Gates: Sources

Bergmann, Seth D. (2018). Section 6.1 and Section 6.2. *Computer Organization with MIPS*. License: CC BY-NC-ND: Attribution-NonCommercial-NoDerivs.

CS301: Computer Architecture. Unit 3. *Saylor Academy*, 2010. License: CC BY: Attribution.

Tarnoff, David. (2020). Episodes 4.01, 4.02, 4.03, 4.04, 4.05, 4.06, 4.07, 4.08, 4.09, 4.10, 5.01, 5.02, 5.03, 6.01, 6.02, 6.03, 6.04, 6.05, 6.06, and 6.07. *Computer Organization and Design Fundamentals Series*. License: CC BY-NC-SA: Attribution-NonCommercial-ShareAlike.