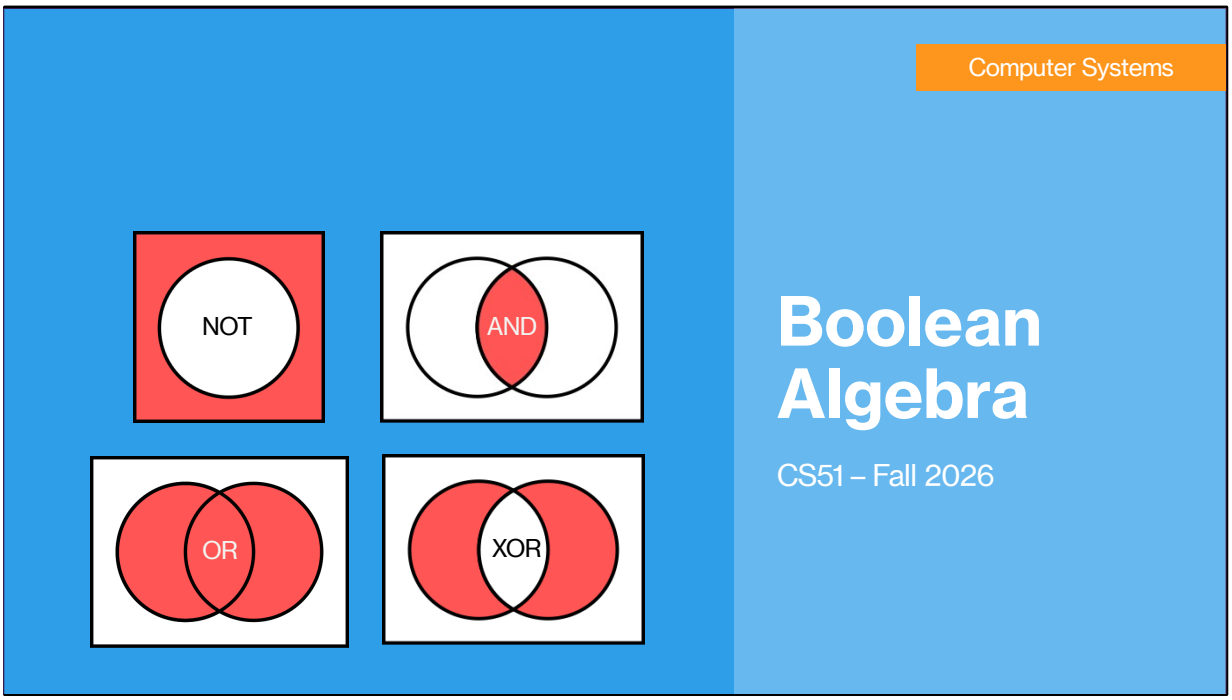


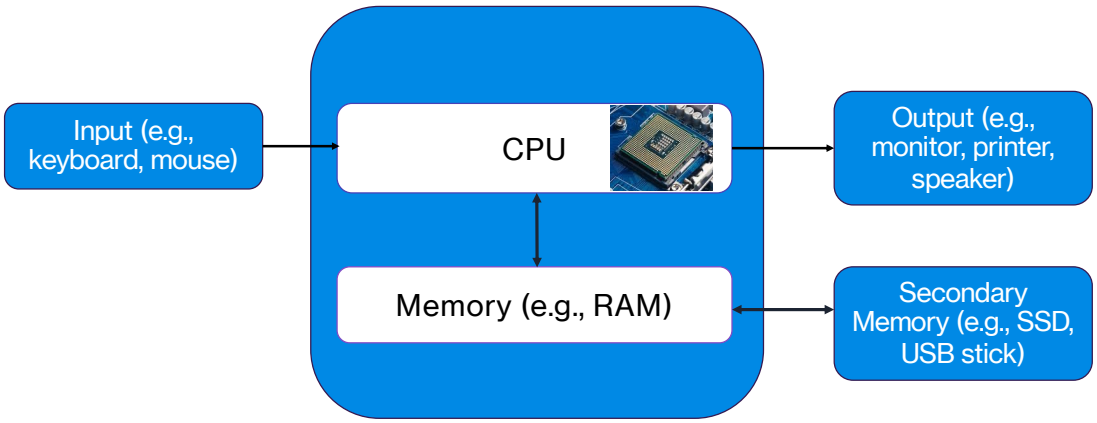
Computer Systems

Boolean Algebra

CS51 – Fall 2026

Today will commence a new unit in our class that will focus on computer systems. We will see how major computer chips are made from physical components like transistors that control two states and thus give rise to a binary representation that enables us to work with a branch of mathematics called Boolean Algebra. We will talk about this concepts both in terms of applying logical operations on abstract 0s and 1s and of how they appear in an advanced programming language like Python.

A simplified view of a computer

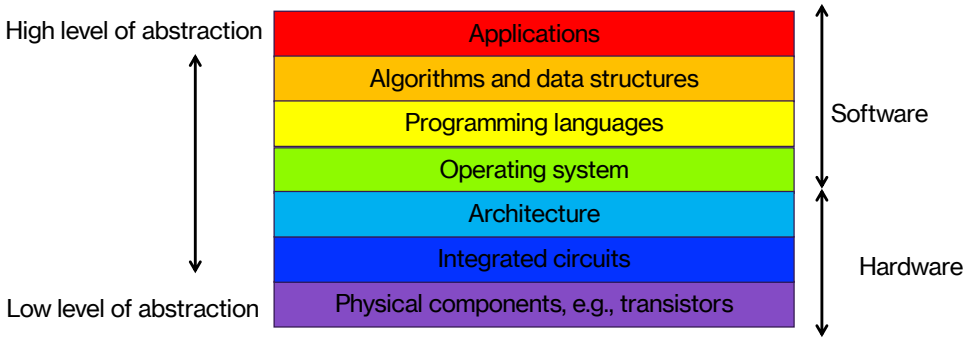


2

A simplified view of a modern computer would show that it comprises mostly of two components. A central processing unit, i.e. the CPU, and memory (also known as primary or main memory). You can think of the CPU as the workhorse of your computer that communicates with the memory and auxiliary devices. The memory stores current data, programs, and instructions that can be accessed directly by the CPU. Any data that should not be changed is stored in a special type of main memory called ROM (read-only memory). For example, the programs that your operating system needs when it boots are stored in ROM. In contrast, RAM (random-access memory) is like our computer's short-term memory for data and programs that need to be read and written but is volatile, that is its information is wiped if power is removed. Secondary memory stores programs and data permanently, without having to worry about what happens when we power off our computer. Secondary memory comes mostly in the form of solid-state drives (SSDs) these days. They are smaller, faster, and less noisy than the older hard-disk drives (HDDs). Other examples of secondary memory are usb drives, memory cards, CDs/DVDs. All this exchange of information can be supplemented by input that the user provides through their keyboard, mouse, etc, and it can be directed to output devices such as the monitor, printers, speakers, etc.

Abstraction

- Critical technique for managing complexity. We will simplify and generalize information by hiding details when they are not important. This allows us to cultivate a higher level of understanding without being overwhelmed with insignificant details.



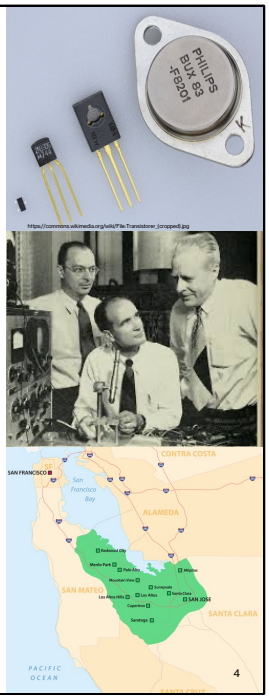
3

It can get pretty easy to get lost in the details of how computers are built and work. So how do we go about it? One of the characteristics that distinguishes computer scientists is a systematic approach to managing complexity by using abstraction. The figure above illustrates levels of abstraction for a computer system. At the highest level of abstraction, we have all applications that our computers offer and users interact with. Think of your favorite browser, music player, photo and video viewing applications, word and spreadsheet processors, etc. Programmers write these applications in code by using algorithms and data structures which they implement in different programming languages, such as Python, Java, Haskell, etc, based on their specific needs. One lower level within the software domain, operating systems interface with the hardware by handling details such as managing the window system, the file manager, and communicating with external devices like a printer or keyboard. Your computer probably runs on one of the operating systems macOS, Windows, or Linux and your phone on iOS or Android. The architecture of a computer offers a way for the hardware and software to communicate through a machine language, a set of low-level instructions that the hardware understands and that the software

translates from language that is relatively easily understood by humans to those instructions. Most computer processors today use an architecture called x86 which was popularized by IBM and Intel. Most phones use the low-power ARM architecture. The physical components of the computer (processor, memory) are built out of integrated circuits (also called chips) which are designed around logic gates. Logic gates are built out of transistors, capacitors, etc and operate on 0s and 1s. Over the next few lectures, we will focus on the lower levels of abstraction of computers and start building our way up.

Transistors

- Transistors are semiconductor devices made out of silicon.
- They act as switches that can be open or closed by applying electricity.
- Integrated circuits in modern computers are built off billions of transistors.
  - They are **small** – you can fit a few thousand of them across the width of a human hair!
  - They are **fast** – they can switch states millions of times per second.
  - They are **reliable** – they can run without any issue for decades.
- Invented in 1947 by Bardeen, Brattain, and Shockley at Bell Lab.
  - They shared the 1956 Nobel Prize in Physics for their achievement.
- A lot of development of transistors and circuits took place in Santa Clara Valley, which by the 1980s started being referred to as the **Silicon Valley**.



At the lowest level of abstraction, we have physical components that are used to make up modern computers. One such type is the transistor. Transistors are semiconductor devices made out of silicon that act as switches that can be open or closed by applying electrical power. Modern computers comprise of integrated circuits made out of billions of transistors! Transistors have a lot of advantages: they are small (as small as 2-3 nanometers!), extremely fast (they can switch between states rapidly), and reliable (they can keep running for decades). Transistors were invented in 1947 by Bardeen, Brattain, and Shockley (infamous for his eugenicist views) at the famous Bell Lab. The three of them shared a Nobel Prize in Physics in 1956 for their invention. Fun fact: Bardeen is the only person to have been awarded the Nobel Prize in Physics twice; the second time was in 1972 with Cooper and Schrieffer for their work on superconductivity. Because a lot of the development of transistors and circuits in general took place in the Santa Clara Valley, California, and because these computer parts are made out of silicon, the region took the name we all know today… Silicon Valley.

Think

- *What is a Boolean (e.g., a Boolean variable)*

5

Let's pause here and think of the following question? What is a boolean e.g., a Boolean variable? Don't worry if you haven't heard the term before.

Think

- *What is a Boolean (e.g., a Boolean variable)*
  - A variable/value that only has two possible values True/1 or False/0

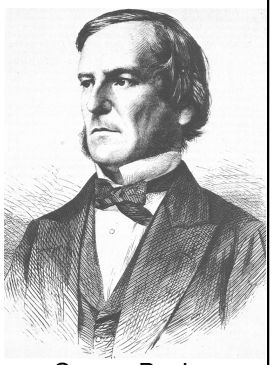
6

If you have prior programming experience, you probably have encountered this concept as two possible values, True (or 1) or False (or 0)

Boolean Algebra

Binary representation and Boolean algebra

- Transistors are powered with electricity and can be in one of two states, allowing us to represent information in **binary**.
- Binary means two states. Common interchangeable representations are:
  - 0 and 1,
  - False and True,
  - Low and High.
- Working only with two states has the advantage of giving us a distinct signal, thus minimizing errors.
- It also works with a well-established branch of Mathematics called **Boolean algebra** that combines Boolean variables.

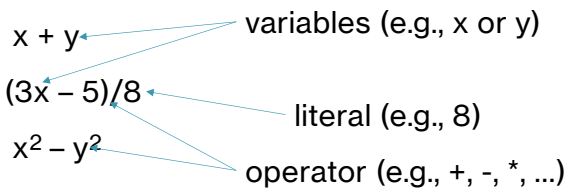


George Boole

8

How does this tie with transistors? Transistors can have one of two states (on or off) which allows us to represent information in binary. Binary means two states. The off state (no electricity flow) is also commonly and interchangeable represented as 0, False, or Low. The on state (electricity flows) is equivalently represented as 1, True, or High. Working only with two states minimizes errors as it gives us a strong signal. It also allows us to work with a well-established branch of Mathematics developed by George Boole, called Boolean algebra. George Boole (who was self-taught) published in 1854 the work *Investigation of the Laws of Thought* which sought to systematically and formally prove logic statements that were true, going beyond Aristotle's philosophical approach to logic. In regular Algebra, variables are numbers and operators perform addition, multiplication, etc. In Boolean algebra, variables are binary and have one of two values, True or False which can be combined using logical operations, such as NOT, AND, and OR that we will cover next.

Algebra as you know it



9

Before we talk about Boolean Algebra, we already know quite a lot through Algebra you saw in school. For example, you are familiar with variables. You have worked with literals (even if you did not know that's what they are called), and you know to combine them using operators.

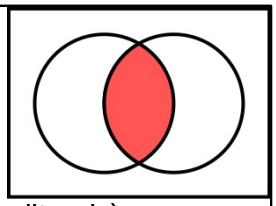
Boolean Algebra

- Boolean algebra is a branch of algebra where variables can only have two values
- Literals: True, False (alternatively: 1, 0)
- Operators?

10

These ideas carry through in Boolean Algebra where now our variables can only have two values. That means the literals can be true/false (or 0/1) only. But what about the operators?

AND operation – Conjunction



- Binary operator, i.e., it is applied two operands (Boolean variables or literals).
- Also denoted as ∧.
- Produces True (1) only if *both* operands are True (1).
- We can use a **truth table** to enumerate all possible values of operands and the effect of the logical operations when applied to them.

X	Y	X AND Y
False	False	False
False	True	False
True	False	False
True	True	True

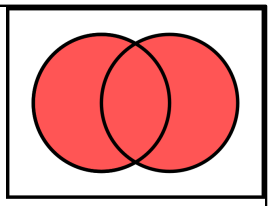
Equivalently

X	Y	X ∧ Y
0	0	0
0	1	0
1	0	0
1	1	1

11

The first operator we will use is called AND. AND, also known as conjunction, is a binary operator, that is it is applied on two operands (Boolean variables or literals). It is also denoted as ∧. AND produces True (1) only if both operands it is applied to are True (1). This relationship can be summarized in a *truth table*. Since AND and ∧ are equivalent, as well as True/1, False/0, you may encounter two commonly used truth tables for AND. Note how only the last row where both X and Y are True/1 produces True/1.

OR operation – Disjunction



- Binary operator, i.e., it is applied two operands. Also denoted as ∨.
- Produces True (1) if *at least* one operand is True (1).
- Truth table for OR:

X	Y	X OR Y
False	False	False
False	True	True
True	False	True
True	True	True

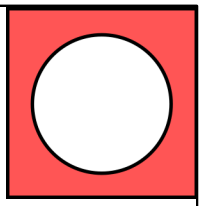
Equivalently

X	Y	X ∨ Y
0	0	0
0	1	1
1	0	1
1	1	1

12

OR also known as disjunction is a binary operator, that is it is applied on two operands. It is also denoted as ∨. OR produces True (1) if at least one of the operands it is applied to is True (1). Similarly to NOT and AND, you can encounter two commonly used truth tables for OR. Note how only the first row where both X and Y are False/0 produces False/0, all other rows produce True/1.

NOT operation – Negation



- Unary operator, i.e., it is applied to only one operand. Also denoted as ¬.
- It flips False (0) to True (1) and, equivalently, True (1) to False (0).
- Truth table for NOT:

X	NOT X
False	True
True	False

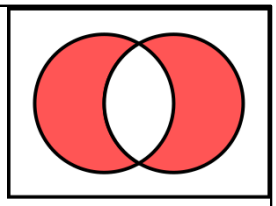
Equivalently

X	¬ X
0	1
1	0

13

We may also want to negate a Boolean variable. NOT also known as Boolean negation is a unary operator, that is it is applied to only one operand. You can also see it represented ¬. What NOT does is that it flips False to True and True to False. Remember, True/False is equivalent to 1/0 so you can think of NOT flipping 0 to 1 and 1 to 0, too. You can see the *truth table* where it flips the state of X to its counterpart. Either truth table is correct depending on how you want to represent the binary values.

XOR operation – Exclusive OR



- Binary operator, i.e., it is applied two operands. Also denoted as $\oplus$.
- Produces True (1) if *exactly* one operand is True (1).
- Truth table for XOR:

X	Y	X XOR Y
False	False	False
False	True	True
True	False	True
True	True	False

Equivalently

X	Y	X $\oplus$ Y
0	0	0
0	1	1
1	0	1
1	1	0

14

Beyond these three basic operators that were introduced by Boole, there have been secondary ones that have been subsequently introduced, with XOR being the most popular. XOR, also known as exclusive OR, is a binary operator, that is it is applied on two operands. It is also denoted as $\oplus$. XOR produces True (1) if exactly one of the operands it is applied to is True (1). Similarly to NOT, AND, OR, you can encounter two commonly used truth tables for XOR. Note how only the second and third row where only one of X and Y is True/1 produces True/1, all other rows produce False/0.

PRACTICE TIME – Evaluate these expressions

- 0 ∨ (¬ 0)
- 1 ∧ (¬ 1)
- ¬ (1 ∧ 0)
- 0 ∧ (¬ 1)
- ¬ (1 ∨ (¬ 1))
- 1 ⊕ (¬ 1)
- (1 ∧ (1 ⊕ ¬(0 ∨ ¬ 1)))

AND

X	Y	X ∧ Y
0	0	0
0	1	0
1	0	0
1	1	1

NOT

X	¬ X
0	1
1	0

OR

X	Y	X ∨ Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

15

Using our understanding of how the NOT, AND, OR, and XOR logical operators work, write what the results of evaluating the following expressions would be.

ANSWER – Evaluate these expressions

- 0 ∨ (¬ 0)= 0 ∨ 1 = 1
- 1 ∧ (¬ 1) = 1 ∧ 0 = 0
- ¬ (1 ∧ 0) = ¬ 0 = 1
- 0 ∧ (¬ 1) = 0 ∧ 0 = 0
- ¬ (1 ∨ (¬ 1)) = ¬ (1 ∨ 0) = ¬ 1 = 0
- 1 ⊕ (¬ 1) = 1 ⊕ 0 = 1
- (1 ∧ (1 ⊕ ¬(0 ∨ ¬ 1))) = (1 ∧ (1 ⊕ ¬(0 ∨ 0))) = (1 ∧ (1 ⊕ ¬0))= (1 ∧ (1 ⊕1))= 1 ∧ 0 = 0

AND

X	Y	X ∧ Y
0	0	0
0	1	0
1	0	0
1	1	1

NOT

X	¬ X
0	1
1	0

OR

X	Y	X ∨ Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR

X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

16

Did you get these?

Boolean functions of three+ variables

- AND($X_1, X_2, \ldots, X_n$) = 0 if any argument is 0, 1 if all arguments are 1.
  - This is equivalent to writing $X_1 \wedge X_2 \wedge \ldots \wedge X_n$
- OR($X_1, X_2, \ldots, X_n$) = 1 if any argument is 1, 0 if all arguments are 0.
  - This is equivalent to writing $X_1 \vee X_2 \vee \cdots \vee X_n$

X	Y	Z	AND(X,Y,Z)	OR(X,Y,Z)
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

17

So far, we have seen that you can apply AND and OR (and XOR) on two arguments. In practice, you can write Boolean functions with more than two variables. For example, AND($X_1, X_2, \ldots, X_n$) would evaluate to 0 if any argument is 0. It would only evaluate to 1 if all arguments were 1. OR($X_1, X_2, \ldots, X_n$) evaluates to 1 as long as least one argument is 1. If all arguments are 0, then it evaluates to 0. You can see that this way, we can write a lot more complex functions.

Think

- 2 + 3 * 5 =
- (2 + 3) * 5 =

18

Going back to basic Algebra, can you evaluate these two expressions?

Operator precedence in Algebra

- 2 + 3 * 5 = 17
- (2 + 3) * 5 = 25

19

Is this what you calculated? In general, we follow the following precedence:
Parentheses
Exponentiation
Multiplication and division
Addition and subtraction

Think

- $1 \vee 1 \wedge 0 =$
- $\neg 0 \wedge 0 =$

20

What if we were to try now in Boolean Algebra?

Operator precedence in Boolean Algebra

- $1 \vee 1 \wedge 0 = 1$

highest precedence (happens first)

- $\neg 0 \wedge 0 = 0$

$\neg$

$\wedge$

$\vee$

lowest precedence (happens later)

21

In general, NOT has the highest precedence followed by AND, and then by OR.

Operator precedence in Boolean Algebra

- $(1 \vee 1) \wedge 0 = 0$
- $\neg(0 \wedge 0) = 1$

highest precedence (happens first)

$\neg$

$\wedge$

$\vee$

lowest precedence (happens later)

22

Note that if we use parentheses, they have again higher precedence.

Think

- $x(y + z) =$

23

Again, let's go back to Algebra, if you saw this expression, what can you do?

Distributive Law in Algebra

- $x(y + z) = xy + xz$

24

Using the distributive law, you can multiply x with each of y and z and add their products.

Think

- $x \wedge (y \vee z) =$

25

Keeping this in mind, what do you think about this Boolean expression?

Distributive Law in Boolean Algebra

- $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
- *How can you check this?*

26

Sounds reasonable? How can you check that these two expressions are equivalent?

Distributive Law in Boolean Algebra

- $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$

x	y	z	y ∨ z	x ∧ (y∨ z)	(x ∧ y)	(x ∧ z)	(x ∧ y) ∨ (x ∧ z)
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

27

You could use truth tables!

PRACTICE TIME – De Morgan's laws

- ¬ (X ∧ Y) = ¬ X ∨ ¬ Y
- ¬ (X ∨ Y) = ¬ X ∧ ¬ Y

X	Y	X ∧ Y	¬ (X ∧ Y)	X ∨ Y	¬ (X ∨ Y)	¬ X	¬ Y	¬ X ∧ ¬ Y	¬ X ∨ ¬ Y
0	0								
0	1								
1	0								
1	1								

28

We can use the truth tables we learned about the different logical operations to evaluate more complex logical expressions. Two such expressions are known as De Morgan's Laws and provide handy equations. The first law states that the negation of X AND Y is the same as the NOT X OR NOT Y. The second law says that NOT (X OR Y) is the same as (NOT X) AND (NOT Y). I have provided you with the four different combinations that X and Y can take. Please fill the rest of the columns, using our understanding of how NOT, AND, and OR work.

ANSWER – De Morgan's laws

- ¬ (X ∧ Y) = ¬ X ∨ ¬ Y
- ¬ (X ∨ Y) = ¬ X ∧ ¬ Y

X	Y	X ∧ Y	¬ (X ∧ Y)	X ∨ Y	¬ (X ∨ Y)	¬ X	¬ Y	¬ X ∧ ¬ Y	¬ X ∨ ¬ Y
0	0	0	1	0	1	1	1	1	1
0	1	0	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	0	1
1	1	1	0	1	0	0	0	0	0

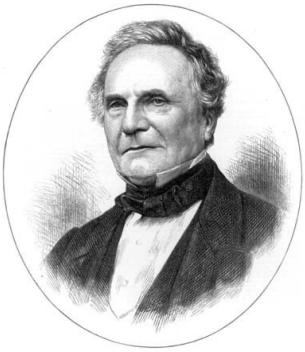
29

As you can see in the green highlighted columns, indeed NOT (X AND Y) matches (NOT X) OR (NOT Y). And in the orange highlighted columns, NOT(X OR Y) matches (NOT X) AND (NOT Y)

A mathematical lineage to the first computer

Augustus De Morgan

Ada Lovelace



Charles Babbage

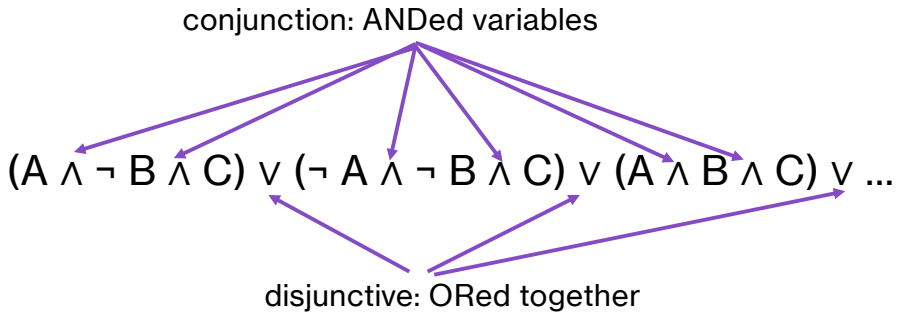
30

Interesting tidbit of history. De Morgan's Laws are named after Augustus De Morgan (1806–1871), a British mathematician and logician who was also the math tutor of Ada Lovelace (1815–1852). Ada Lovelace was an English mathematician and the only legitimate child of poet Lord Byron! She is known as the first computer programmer and she maintained correspondence and translated the work of English mathematician and inventor Charles Babbage (1791-1871) who worked on the idea of what is considered the first digital computer, the Analytical Engine, which would be a general-purpose, fully program-controlled computer.

DNFs and CNFs

Disjunctive normal form (DNF)

- **Theorem (Boole, 1847):** Any Boolean function can be represented as a disjunction (ORs) of conjunctions (ANDs) of its arguments and their negation (NOT). This is also known as **disjunctive normal form** (DNF).



32

Along with his Algebra, Boole wrote a very interesting theorem in 1847. Any Boolean function can be represented as a disjunction of conjunctions (one OR of a bunch of ANDs) of its arguments and their negation (NOT). This is also known as **disjunctive normal form** (DNF) or a sum of products/minterms.

Disjunctive normal form (DNF)

One way this can happen is by using “laws”, specifically:

$$
\begin{array}{rcll}
(\neg\neg x) & \rightsquigarrow & x & \text{double negation} \\
(\neg(x \vee y)) & \rightsquigarrow & ((\neg x) \wedge (\neg y)) & \text{De Morgan's laws} \\
(\neg(x \wedge y)) & \rightsquigarrow & ((\neg x) \vee (\neg y)) & \\
(x \wedge (y \vee z)) & \rightsquigarrow & ((x \wedge y) \vee (x \wedge z)) & \text{distributive law} \\
((x \vee y) \wedge z) & \rightsquigarrow & ((x \wedge z) \vee (y \wedge z)) &
\end{array}
$$

33

How can this be? We could use lots of manipulations, such as De Morgan's laws, distributive law etc

Minterms

Boolean expressions consisting of the conjunction (AND) of all variables of a Boolean function. Each variable will appear exactly once in either its true (if the variable was 1) or complemented form (if the variable was 0). The minterm is formed using the values of the variables which make the value of the minterm equal to 1.

Row number	X	Y	Z	Minterm
0	0	0	0	$m_0 = \neg X \wedge \neg Y \wedge \neg Z$
1	0	0	1	$m_1 = \neg X \wedge \neg Y \wedge Z$
2	0	1	0	$m_2 = \neg X \wedge Y \wedge \neg Z$
3	0	1	1	$m_3 = \neg X \wedge Y \wedge Z$
4	1	0	0	$m_4 = X \wedge \neg Y \wedge \neg Z$
5	1	0	1	$m_5 = X \wedge \neg Y \wedge Z$
6	1	1	0	$m_6 = X \wedge Y \wedge \neg Z$
7	1	1	1	$m_7 = X \wedge Y \wedge Z$

34

Or we could do this directly from the truth table by creating minterms. A minterm is a Boolean logical expression consisting of the conjunction (AND) of all variables, each appearing exactly once in either its true (if the variable was 1) or complemented form (if the variable was 0). The minterm is formed using the values of the variables which make the value of the minterm equal to 1. We symbolize minterms with small m subscripted from 0 to to (2^n)-1, where n is the number of the variables of the Boolean function. For example, here, m_0 corresponds to the 0^{th} row where all three variables are false, as such the minterm is going to be not X and not Y and not Z.

Minterms

Boolean expressions consisting of the conjunction (AND) of all variables of a Boolean function. Each variable will appear exactly once in either its true (if the variable was 1) or complemented form (if the variable was 0). The minterm is formed using the values of the variables which make the value of the minterm equal to 1.

Row number	X	Y	Z	$m_3 = \neg X \wedge Y \wedge Z$
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	**0**	**1**	**1**	**1**
4	1	0	0	0
5	1	0	1	0
6	1	1	0	0
7	1	1	1	0

35

Note that each minterm will be True/1 for exactly one entry in the truth table, e.g. m3 will be true only when x=0, y=1,z=1.

Example – MAJ function

- MAJ($X_1, X_2, \ldots, X_n$) = 1 if (strictly) more arguments are 1 than 0, 0 otherwise.

X	Y	Z	MAJ(X,Y,Z)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

36

Let's see how we could calculate a DNF for a function called MAJ. MAJ over n variables will return 1 if strictly more arguments are 1 than 0. This is its truth table if we were to apply it on three variables. We will next figure out a way to write a DNF that will capture the entire truth table.

Example – DNF of MAJ function

- MAJ($X_1, X_2, \ldots, X_n$) = 1 if (strictly) more arguments are 1 than 0, 0 otherwise.

X	Y	Z	MAJ(X,Y,Z)	Minterm
0	0	0	0	$m_0 = \neg X \wedge \neg Y \wedge \neg Z$
0	0	1	0	$m_1 = \neg X \wedge \neg Y \wedge Z$
0	1	0	0	$m_2 = \neg X \wedge Y \wedge \neg Z$
0	**1**	**1**	**1**	$\mathbf{m_3 = \neg X \wedge Y \wedge Z}$
1	0	0	0	$m_4 = X \wedge \neg Y \wedge \neg Z$
1	**0**	**1**	**1**	$\mathbf{m_5 = X \wedge \neg Y \wedge Z}$
1	**1**	**0**	**1**	$\mathbf{m_6 = X \wedge Y \wedge \neg Z}$
1	**1**	**1**	**1**	$\mathbf{m_7 = X \wedge Y \wedge Z}$

OR these together

- Isolate the rows where the output is equal to 1 and then connect with a disjunction (OR) the corresponding minterms.

37

To calculate its DNF, we will isolate the rows where the output is equal to 1 and then connect the minterms using a disjunction, i.e. OR

Example – DNF of MAJ function

- MAJ($X_1, X_2, \ldots, X_n$) = 1 if (strictly) more arguments are 1 than 0, 0 otherwise.

X	Y	Z	MAJ(X,Y,Z)	Minterm
0	0	0	0	$m_0 = \neg X \wedge \neg Y \wedge \neg Z$
0	0	1	0	$m_1 = \neg X \wedge \neg Y \wedge Z$
0	1	0	0	$m_2 = \neg X \wedge Y \wedge \neg Z$
0	**1**	**1**	**1**	$\mathbf{m_3 = \neg X \wedge Y \wedge Z}$
1	0	0	0	$m_4 = X \wedge \neg Y \wedge \neg Z$
1	**0**	**1**	**1**	$\mathbf{m_5 = X \wedge \neg Y \wedge Z}$
1	**1**	**0**	**1**	$\mathbf{m_6 = X \wedge Y \wedge \neg Z}$
1	**1**	**1**	**1**	$\mathbf{m_7 = X \wedge Y \wedge Z}$

OR these together

- Isolate the rows where the output is equal to 1 and then connect with a disjunction (OR) the corresponding minterms.
- MAJ(X, Y, Z) = $m_3 \vee m_5 \vee m_6 \vee m_7$ = (¬X ∧ Y ∧ Z) ∨ (X ∧ ¬Y ∧ Z) ∨ (X ∧ Y ∧ ¬Z) ∨ (X ∧ Y ∧ Z)

38

This results in MAJ being able to be described as (¬X ∧ Y ∧ Z) ∨ (X ∧ ¬Y ∧ Z) ∨ (X ∧ Y ∧ ¬Z) ∨ (X ∧ Y ∧ Z)

PRACTICE TIME – DNF of ODD function

ODD($X_1, X_2, \ldots, X_n$) = 1 if an odd number of arguments is 1, 0 otherwise

X	Y	Z	ODD(X,Y,Z)	Minterm
0	0	0	0	$m_0 =$
0	0	1	1	$m_1 =$
0	1	0	1	$m_2 =$
0	1	1	0	$m_3 =$
1	0	0	1	$m_4 =$
1	0	1	0	$m_5 =$
1	1	0	0	$m_6 =$
1	1	1	1	$m_7 =$

ODD(X, Y, Z) =

39

Your turn now. Can you calculate the DNF of the ODD function over three arguments? The ODD function returns 1 if an odd number of arguments is 1, and 0 otherwise. Start by calculating the minterms and then combine them in the way we saw.

PRACTICE TIME – DNF of ODD function

ODD($X_1, X_2, \ldots, X_n$) = 1 if an odd number of arguments is 1, 0 otherwise

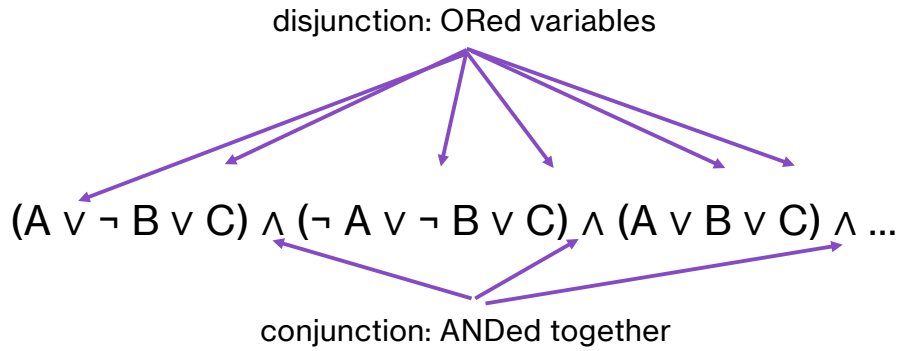
X	Y	Z	ODD(X,Y,Z)	Minterm
0	0	0	0	$m_0 = \neg X \wedge \neg Y \wedge \neg Z$
0	**0**	**1**	**1**	$\boldsymbol{m_1 = \neg X \wedge \neg Y \wedge Z}$
0	**1**	**0**	**1**	$\boldsymbol{m_2 = \neg X \wedge Y \wedge \neg Z}$
0	1	1	0	$m_3 = \neg X \wedge Y \wedge Z$
1	**0**	**0**	**1**	$\boldsymbol{m_4 = X \wedge \neg Y \wedge \neg Z}$
1	0	1	0	$m_5 = X \wedge \neg Y \wedge Z$
1	1	0	0	$m_6 = X \wedge Y \wedge \neg Z$
1	**1**	**1**	**1**	$\boldsymbol{m_7 = X \wedge Y \wedge Z}$

ODD(X, Y, Z) = $m_1 \vee m_2 \vee m_4 \vee m_7$ = (¬X ∧ ¬Y ∧ Z) ∨ (¬X ∧ Y ∧ ¬Z) ∨ (X ∧ ¬Y ∧ ¬Z) ∨ (X ∧ Y ∧ Z)

40

Conjunctive normal form (CNF)

- **Theorem (Boole, 1847):** Any Boolean function can be represented as a conjunction (AND)s of disjunction (ORs) of its arguments and their negation (NOT). This is also known as **conjunctive normal form** (CNF).



41

Remarkably, there is an equivalent theorem that says that we can represent any Boolean function as a conjunction of disjunction. This is known as the conjunctive normal form or CNF.

Maxterms

Boolean expressions consisting of the disjunction (OR) of all variables of a Boolean function. Each variable will appear exactly once in either its true (if the variable was 0) or complemented form (if the variable was 1). The maxterm is formed using the values of the variables which make the value of the maxterm equal to 0.

Row number	X	Y	Z	Minterm	Maxterm
0	0	0	0	$m_0 = \neg X \wedge \neg Y \wedge \neg Z$	$M_0 = X \vee Y \vee Z$
1	0	0	1	$m_1 = \neg X \wedge \neg Y \wedge Z$	$M_1 = X \vee Y \vee \neg Z$
2	0	1	0	$m_2 = \neg X \wedge Y \wedge \neg Z$	$M_2 = X \vee \neg Y \vee Z$
3	0	1	1	$m_3 = \neg X \wedge Y \wedge Z$	$M_3 = X \vee \neg Y \vee \neg Z$
4	1	0	0	$m_4 = X \wedge \neg Y \wedge \neg Z$	$M_4 = \neg X \vee Y \vee Z$
5	1	0	1	$m_5 = X \wedge \neg Y \wedge Z$	$M_5 = \neg X \vee Y \vee \neg Z$
6	1	1	0	$m_6 = X \wedge Y \wedge \neg Z$	$M_6 = \neg X \vee \neg Y \vee Z$
7	1	1	1	$m_7 = X \wedge Y \wedge Z$	$M_7 = \neg X \vee \neg Y \vee \neg Z$

42

This time, we will use maxterms. A maxterm is a Boolean expression consisting of the disjunction (OR) of all variables of a Boolean function. Each variable will appear exactly once in either its true (if the variable was 0) or complemented form (if the variable was 1). The maxterm is formed using the values of the variables which make the value of the maxterm equal to 0. We symbolize maxterms with capital M subscripted from 0 to to (2^n)-1, where n is the number of the variables of the Boolean function. For example, here, M_0 corresponds to the 0^{th} row where all three variables are false, as such the maxterm is going to be X or Y or Z.

Maxterms

Boolean expressions consisting of the disjunction (OR) of all variables of a Boolean function. Each variable will appear exactly once in either its true (if the variable was 0) or complemented form (if the variable was 1). The maxterm is formed using the values of the variables which make the value of the maxterm equal to 0.

Row number	X	Y	Z	$M_5 = \neg X \vee Y \vee \neg Z$
0	0	0	0	1
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	1
5	**1**	**0**	**1**	**0**
6	1	1	0	1
7	1	1	1	1

43

Note that each maxterm will be False/0 for exactly one entry in the truth table, e.g. M5 will be False only when x=1, y=0,z=1.

Example – CNF of MAJ function

- MAJ($X_1, X_2, \ldots, X_n$) = 1 if (strictly) more arguments are 1 than 0, 0 otherwise.

X	Y	Z	MAJ(X,Y,Z)	Maxterm
0	**0**	**0**	**0**	$\boldsymbol{M_0 = X \vee Y \vee Z}$
0	**0**	**1**	**0**	$\boldsymbol{M_1 = X \vee Y \vee \neg Z}$
0	**1**	**0**	**0**	$\boldsymbol{M_2 = X \vee \neg Y \vee Z}$
0	1	1	1	$M_3 = X \vee \neg Y \vee \neg Z$
1	**0**	**0**	**0**	$\boldsymbol{M_4 = \neg X \vee Y \vee Z}$
1	0	1	1	$M_5 = \neg X \vee Y \vee \neg Z$
1	1	0	1	$M_6 = \neg X \vee \neg Y \vee Z$
1	1	1	1	$M_7 = \neg X \vee \neg Y \vee \neg Z$

AND these together

- Isolate the rows where the output is equal to 0 and then connect with a conjunction (AND) the corresponding maxterms.
- MAJ(X, Y, Z) = $M_0 \wedge M_1 \wedge M_2 \wedge M_4$ = (X ∨ Y ∨ Z) ∧ (X ∨ Y ∨ ¬Z) ∧ (X ∨ ¬Y ∨ Z) ∧ (¬X ∨ Y ∨ Z)

44

Let's go back to our example of the function MAJ over three variables. To find the CNF, we will isolate the rows where the output is equal to 0 and then connect with ANDs the corresponding maxterms. Do you see the equivalence with DNFs? In DNFs we OR the minterms for rows that are 1. In CNFs, we AND the maxterms for rows that are 0.

PRACTICE TIME – CNF of ODD function

ODD($X_1, X_2, \ldots, X_n$) = 1 if an odd number of arguments is 1, 0 otherwise

X	Y	Z	ODD(X,Y,Z)	Maxterm
0	0	0	0	$M_0 =$
0	0	1	1	$M_1 =$
0	1	0	1	$M_2 =$
0	1	1	0	$M_3 =$
1	0	0	1	M_4
1	0	1	0	$M_5 =$
1	1	0	0	$M_6 =$
1	1	1	1	$M_7 =$

ODD(X, Y, Z) =

45

Your turn now. Can you calculate the CNF of the ODD function over three arguments? The ODD function returns 1 if an odd number of arguments is 1, and 0 otherwise. Start by calculating the maxterms and then combine them in the way we saw.

ANSWER – CNF of ODD function

ODD($X_1, X_2, \ldots, X_n$) = 1 if an odd number of arguments is 1, 0 otherwise

X	Y	Z	ODD(X,Y,Z)	Maxterm
0	0	0	0	$M_0 = X \vee Y \vee Z$
0	0	1	1	$M_1 = X \vee Y \vee \neg Z$
0	1	0	1	$M_2 = X \vee \neg Y \vee Z$
0	1	1	0	$M_3 = X \vee \neg Y \vee \neg Z$
1	0	0	1	$M_4 = \neg X \vee Y \vee Z$
1	0	1	0	$M_5 = \neg X \vee Y \vee \neg Z$
1	1	0	0	$M_6 = \neg X \vee \neg Y \vee Z$
1	1	1	1	$M_7 = \neg X \vee \neg Y \vee \neg Z$

ODD(X, Y, Z) $= M_0 \wedge M_3 \wedge M_5 \wedge M_6 =$ (X ∨ Y ∨ Z) ∧ (X ∨ ¬Y ∨ ¬Z) ∧ (¬X ∨ Y ∨ ¬Z) ∧ (¬X ∨ ¬Y ∨ Z)

46

If you isolated the rows where the ODD function returns 0, and connected with ANDs those corresponding maxterms, then you successfully calculated the CNF of the ODD function over three arguments.

Gates

47

CPU: central processing unit

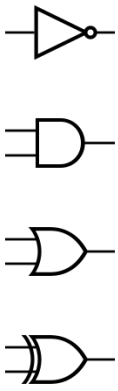
CPU

48

So how are all the things we saw today connected with what we started with? The idea that we would be working at the system level for the next few class meetings? Remember, we said that in your computer, there is the central processing unit or CPU.

Inside a CPU

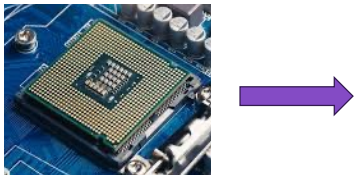
Gates



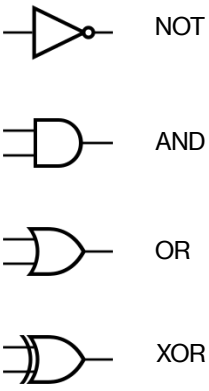
49

The CPU is made out of gazillion gates (which consist of transistors).

Inside a CPU



Gates

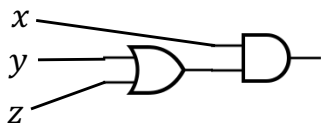


50

Gates receive input in the form of electricity through wires and then pass (or not) that as output. In fact, all the logical connectives we saw today, can be translated to gates. For example, there is a NOT gate called an inverter, where it reads the signal of one input. If the voltage is LOW (0), then the output is the opposite, HIGH (1). If voltage is HIGH (1), then the output is LOW (0). All the other logical operators we saw were binary which means that their equivalent gates receive two input wires and give as an output a single wire

Gates

$$x \wedge (y \vee z) =$$

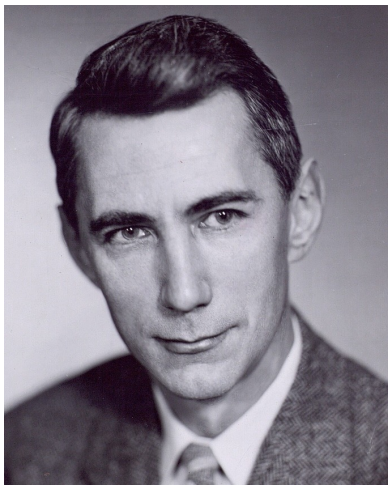


51

That means there is a direct translation from Boolean Algebra to logical gates and a Boolean expression like this can be modeled as a circuit!

From Boolean Algebra to Circuits

- In 1937, **Claude Shannon** translated Boolean logic into a physical format with electronics.
- This work became the foundation of **circuit design** and made it possible to design the computers we know today.
- Shannon also introduced many of the core ideas of abstraction, encoding, and compression we use today.



Claude Shannon

52

The person that translated Boolean logic into a physical format is Claude Shannon (1916-2001) in his famous Master's Thesis, "A Symbolic Analysis of Relay and Switching Circuits". This work became the foundation of circuit design and made it possible to design the computers we know today (we will see examples of such circuits in a few weeks). Shannon also introduced many of the core ideas of abstraction, encoding, and compression we use today and he is considered the father of information theory.

Practice Problems

Practice Problems – Problem 1

- You are given the following truth table for a Boolean function with three variables, A, B, and C, and an output column F.
- What are the disjunctive and conjunctive normal forms based on this truth table?

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

54

Here are two practice problems for you to work at home. Give them an honest try before turning to the answers

Practice Problems – Answer 1

- Disjunctive normal form:
  $F = (\neg A \wedge \neg B \wedge \neg C) \vee (\neg A \wedge B \wedge \neg C) \vee (A \wedge \neg B \wedge C) \vee (A \wedge B \wedge \neg C)$
- Conjunctive normal form:
  $F = (A \vee B \vee \neg C) \wedge (A \vee \neg B \vee \neg C) \wedge (\neg A \vee B \vee C) \wedge (\neg A \vee \neg B \vee \neg C)$

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

55