MATH 8

LOGIC II: MORE LOGIC OPERATIONS — FULL 2026–27

Introduction

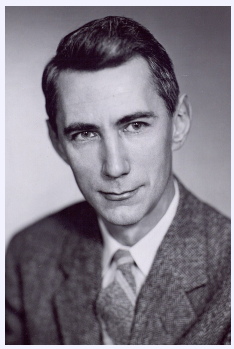
In the previous handout we built our logic vocabulary: propositions, NOT, AND, OR, truth tables, and the basic logic laws, including De Morgan's laws. This handout does three new things with that vocabulary.

- We give names to the two extreme kinds of formulas: **tautologies** (always true) and **contradictions** (always false).
- We meet four more operations — **XOR, NAND, NOR, XNOR** — and discover something surprising: NAND alone (or NOR alone) is enough to build *every* logic operation there is.
- We see how all of this becomes *physical*: electronic **logic gates**, a circuit that adds binary numbers, and a circuit that remembers a bit.

In the next handout we turn to "if . . . then . . ." statements, and later handouts use all of these tools to study mathematical proof.

Historical Note

In 1937 a 21-year-old MIT graduate student named **Claude Shannon** wrote a master's thesis titled *A Symbolic Analysis of Relay and Switching Circuits*. Its big idea fits in one sentence: *every circuit of switches can be described by a Boolean formula, and every Boolean formula can be built as a circuit of switches.* (Two switches one after the other — "in series" — let current through only if both are closed: that's AND. Two switches side by side — "in parallel" — let current through if at least one is closed: that's OR.) With this idea, the logic of George Boole, whom we met in the previous handout, became the design language of digital electronics. The thesis even contained a design for a circuit that adds binary numbers, like the adders in this handout. It has been called the most important master's thesis of the 20th century.



Claude Shannon (1916–2001)

In 1948 Shannon went on to found **information theory**, the mathematics behind all digital communication, which made the word *bit* (binary digit) famous. He was also a playful inventor: he built juggling machines and a mechanical mouse that learned its way through a maze, and he was known for riding a unicycle through the halls of Bell Labs.

Review: Basic Operations

Recall our three basic logic operations:

A	B	$\neg A$	$A \wedge B$	$A \vee B$
T	T	F	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	F

We will also keep using the logic laws from the previous handout (commutative, associative, distributive, double negation, identity, domination, and De Morgan's laws). Today we'll explore additional operations that are built from these basics but are important enough to have their own names.

Tautologies and Contradictions

In the previous handout we checked equivalences by computing truth tables and noticing that two columns match. Two extreme kinds of formula pop up over and over as we do this — the ones whose entire output column is T, and the ones whose entire output column is F. They deserve names.

Tautology and Contradiction

A **tautology** is a formula that is *always true*, whatever the truth values of its variables.
A **contradiction** is a formula that is *always false*, whatever the truth values of its variables.

Example 1. $A \vee \neg A$ is a tautology (called the **Law of Excluded Middle**):

A	$\neg A$	$A \vee \neg A$
T	F	T
F	T	T

Either A is true or A is false — there's no middle ground!

Example 2. $A \wedge \neg A$ is a contradiction:

A	$\neg A$	$A \wedge \neg A$
T	F	F
F	T	F

A cannot be both true and false at the same time!

Most formulas are *neither*: they are true in some rows and false in others. To show that a formula is *not* a tautology, one row where it is false is enough; to show that it *is* a tautology, you must check every row (or use the logic laws).

Quick Check

1. Is $A \vee \neg A \vee B$ a tautology? Explain without a truth table.
2. Is $\neg(A \vee \neg A)$ a tautology, a contradiction, or neither?
3. Is $(A \wedge B) \vee \neg A$ a tautology? If not, find a row where it is false.

XOR: Exclusive Or

In everyday speech, "or" often means "one or the other, but not both." For example, "Would you like tea or coffee?" usually means you can have one, not both. In the previous handout we decided that in logic, OR is always *inclusive*. The "not both" version gets its own name: **exclusive or**, abbreviated **XOR**.

XOR (Exclusive Or)

$A \text{ XOR } B$ (also written $A \oplus B$) is true when *exactly one* of A and B is true.

A	B	$A \vee B$ (inclusive)	$A \text{ XOR } B$ (exclusive)
T	T	T	F
T	F	T	T
F	T	T	T
F	F	F	F

Notice that XOR differs from OR only in the first row: when both are true, XOR is false. Another way to say it: *$A \text{ XOR } B$ is true exactly when A and B are different.*

Expressing XOR with Basic Operations

"Exactly one" means "at least one, but not both." Translating word by word:

$$A \text{ XOR } B \equiv (A \vee B) \wedge \neg(A \wedge B).$$

"Exactly one" also means "A is true and B is false, or A is false and B is true":

$$A \text{ XOR } B \equiv (A \wedge \neg B) \vee (\neg A \wedge B).$$

Properties of XOR

- $A \text{ XOR } A \equiv F$ (anything XOR itself is false)
- $A \text{ XOR } F \equiv A$ (XOR with false doesn't change the value)
- $A \text{ XOR } T \equiv \neg A$ (XOR with true flips the value)
- $A \text{ XOR } B \equiv B \text{ XOR } A$ (commutative)
- $(A \text{ XOR } B) \text{ XOR } C \equiv A \text{ XOR } (B \text{ XOR } C)$ (associative; check it with an 8-row truth table)

Because XOR is associative, a long chain $A_1 \text{ XOR } A_2 \text{ XOR } \cdots \text{ XOR } A_n$ needs no parentheses, and there is a neat rule for its value. Start with F and go through the inputs one at a time: XOR with an F input keeps the running value, and XOR with a T input flips it. So the final answer is T exactly when we flipped an odd number of times:

A chain of XORs is true exactly when an odd number of its inputs are true.

Quick Check

4. Evaluate $(T \text{ XOR } F) \text{ XOR } T$.
5. If A is true and B is true, what is $A \text{ XOR } B$?
6. Evaluate $T \text{ XOR } F \text{ XOR } T \text{ XOR } T \text{ XOR } F$ in one step, using the odd-number rule.

NAND: Not And

NAND (Not And)

NAND is the negation of AND: $A \text{ NAND } B \equiv \neg(A \wedge B)$.

A	B	$A \wedge B$	$A \text{ NAND } B$
T	T	T	F
T	F	F	T
F	T	F	T
F	F	F	T

NAND is false only when both inputs are true. Equivalently (by De Morgan), it is true as soon as at least one input is false: $A \text{ NAND } B \equiv \neg A \vee \neg B$.

NAND is Universal

Here's a remarkable fact: **NAND alone can express every logic operation!** This makes NAND extremely important in computer chip design. Let's see why it is true. The argument has two steps.

Step 1: Every truth table can be written with $\neg$, $\wedge$, $\vee$

Take any truth table. Look at the rows where the output is T. For each such row, write an AND that is true in *that row only*: for example, the row $A = T$, $B = F$ gives $A \wedge \neg B$. Then join all these ANDs with OR. The result is true exactly in the chosen rows — so it has exactly the truth table we wanted.

Example. XOR is true in the rows (T, F) and (F, T). The recipe gives $(A \wedge \neg B) \vee (\neg A \wedge B)$ — exactly our second formula for XOR.

(If the output column has no T at all, use the contradiction $A \wedge \neg A$.)

Step 2: NOT, AND, OR can each be built from NAND.

NOT from NAND

$$\neg A \equiv A \text{ NAND } A$$

Check: if $A = T$, then $A \text{ NAND } A = \neg(T \wedge T) = \neg T = F$. If $A = F$, then $A \text{ NAND } A = \neg(F \wedge F) = \neg F = T$. In both cases we get $\neg A$. ✓

AND from NAND

$$A \wedge B \equiv (A \text{ NAND } B) \text{ NAND } (A \text{ NAND } B)$$

This works because $A \text{ NAND } B = \neg(A \wedge B)$, and NANDing something with itself is NOT (previous box). So we get $\neg(\neg(A \wedge B)) = A \wedge B$.

OR from NAND

$$A \vee B \equiv (A \text{ NAND } A) \text{ NAND } (B \text{ NAND } B)$$

This works because by De Morgan's law, $A \vee B \equiv \neg(\neg A \wedge \neg B)$. We have $\neg A = A \text{ NAND } A$ and $\neg B = B \text{ NAND } B$, and then we need the NOT of their AND — which is exactly a NAND!

Putting the two steps together: write any operation with $\neg, \wedge, \vee$ (Step 1), then replace each $\neg$, $\wedge$, $\vee$ by its NAND version (Step 2). Counting gates: NOT costs 1 NAND gate, AND costs 2, and OR costs 3.

NOR: Not Or

NOR (Not Or)

NOR is the negation of OR: $A \text{ NOR } B \equiv \neg(A \vee B)$.

A	B	$A \vee B$	$A \text{ NOR } B$
T	T	T	F
T	F	T	F
F	T	T	F
F	F	F	T

NOR is true only when *both* inputs are false: $A \text{ NOR } B$ means "neither A nor B," and by De Morgan $A \text{ NOR } B \equiv \neg A \wedge \neg B$.

Like NAND, NOR is also universal — any logic operation can be built using only NOR. The argument is the same two steps: Step 1 is unchanged, and for Step 2 we build NOT, OR, and AND from NOR. Here is NOT; you'll build OR in the Quick Check below and AND in the homework.

NOT from NOR

$$\neg A \equiv A \text{ NOR } A$$

If $A = T$: $A \text{ NOR } A = \neg(T \vee T) = \neg T = F$. If $A = F$: $A \text{ NOR } A = \neg(F \vee F) = \neg F = T$. ✓

Quick Check

7. What is $T \text{ NAND } F$?
8. What is $T \text{ NOR } T$?
9. Express OR using only NOR operations. (*Hint:* OR is the NOT of NOR.)

XNOR: Exclusive Nor (Equivalence)

XNOR (Exclusive Nor)

XNOR is the negation of XOR: $A \text{ XNOR } B \equiv \neg(A \text{ XOR } B)$.

A	B	$A \text{ XOR } B$	$A \text{ XNOR } B$
T	T	F	T
T	F	T	F
F	T	T	F
F	F	F	T

XNOR is true exactly when A and B have the *same* truth value, so it is also called **equivalence**. In the next handout this operation comes back under a third name, "A if and only if B." (Careful: XNOR is *not* "neither A nor B" — that is NOR.)

Expressing XNOR with Basic Operations

"Same value" means "both true or both false":

$$A \text{ XNOR } B \equiv (A \wedge B) \vee (\neg A \wedge \neg B).$$

We can also negate our second formula for XOR, using De Morgan twice:

$$A \text{ XNOR } B \equiv \neg\big((A \wedge \neg B) \vee (\neg A \wedge B)\big) \equiv \neg(A \wedge \neg B) \wedge \neg(\neg A \wedge B) \equiv (\neg A \vee B) \wedge (A \vee \neg B).$$

Properties of XNOR

- $A \text{ XNOR } A \equiv T$ (always agrees with itself)
- $A \text{ XNOR } T \equiv A$ (XNOR with true returns the input)
- $A \text{ XNOR } F \equiv \neg A$ (XNOR with false flips the input)

- $A \text{ XNOR } B \equiv B \text{ XNOR } A$ (commutative)
- $(A \text{ XNOR } B) \text{ XNOR } C \equiv A \text{ XNOR } (B \text{ XNOR } C)$ (associative)

XNOR and Tautologies

Two formulas X and Y are equivalent exactly when their columns agree in *every* row of the truth table. The column of $X \text{ XNOR } Y$ is T exactly in the rows where X and Y agree. So:

$$X \equiv Y \quad \textbf{exactly when} \quad X \text{ XNOR } Y \textbf{ is a tautology.}$$

Every logic law from the previous handout is secretly a tautology! For instance, De Morgan's law says that $\neg(A \vee B) \text{ XNOR } (\neg A \wedge \neg B)$ is always true.

Quick Check

10. Evaluate $(T \text{ XNOR } F) \text{ XNOR } T$.
11. For which truth values of A and B is $A \text{ XNOR } B$ equal to $\neg A$?
12. Explain without a truth table why $\neg(A \wedge B) \text{ XNOR } (\neg A \vee \neg B)$ is a tautology.

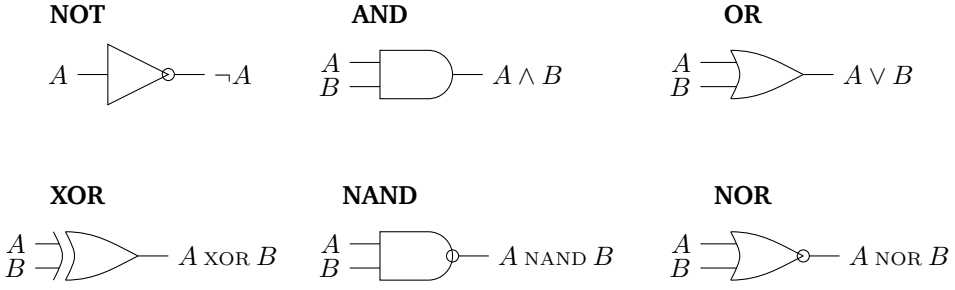
Logic Gates in Electronics

We've spent this handout (and the last one) treating logical operations as pure mathematics — symbols on paper, truth tables, algebraic identities. Here is the punchline: every one of these operations corresponds to a physical electrical circuit that engineers can build out of transistors. The whole digital world — every computer, every phone, every microchip — is a vast network of such circuits. The next few pages take a quick tour of how this works, and then build two essential circuits: a binary *adder* and a *memory cell*.

In electronics, logical operations are carried out by devices called **logic gates**: circuits that take electrical signals as inputs and produce an output according to a logic rule. Each wire carries either a LOW voltage (near 0 volts) or a HIGH voltage (for example, 5 volts in older chips). Engineers write **1** for HIGH, meaning T, and **0** for LOW, meaning F. From now on, in circuits, we will do the same.

Standard Gate Symbols

Each logic operation has a standard symbol used in circuit diagrams:



The small circle on the NOT, NAND, and NOR gates indicates negation. (An XNOR gate is drawn as an XOR gate with a small circle at its output.)

Try it online

You can build and test logic circuits in your browser at https://logic.ly/demo/ (free demo — no account needed). Drag gates onto the canvas, wire them together, flip the input switches, and watch the output lamps light up. It's the fastest way to check whether a circuit you designed actually works.

Why NAND and NOR Gates are Special

Computer chips are built from millions or billions of tiny transistors arranged as logic gates. NAND and NOR gates are among the simplest and cheapest gates to build from transistors, and since each of them can create any other operation, chip designers use them everywhere.

A famous example is the 7400 chip, which contains four NAND gates and has been used in countless electronic devices since the 1960s. An even more dramatic example: the Apollo Guidance Computer, which helped fly astronauts to the Moon in 1969, was built almost entirely from one kind of chip, each holding two NOR gates — about 5,600 NOR gates in all.

Binary Addition with Logic Gates

One practical application of logic gates is arithmetic. Let's see how to add two one-digit binary numbers (bits) using logic gates.

Historical Note: An Adder on the Kitchen Table

In November 1937, the same year as Shannon's thesis, **George Stibitz**, a mathematician at Bell Telephone Laboratories, took a few telephone relays home and built a circuit on his kitchen table that added two binary digits, using flashlight bulbs to show the answer and metal strips cut from a tobacco tin as input keys. It became known as the "Model K" (K for *kitchen*). It grew into a relay calculator that Stibitz operated in 1940 over a telephone line from a mathematics meeting in New Hampshire, more than 200 miles from the machine in New York — one of the first uses of a computer from far away. Within a few years, electronic computers such as ENIAC (1945) were doing arithmetic with about 18,000 vacuum tubes.

Half Adder

When we add two 1-bit binary numbers A and B, we get:

- $0 + 0 = 0$ (no carry)
- $0 + 1 = 1$ (no carry)
- $1 + 0 = 1$ (no carry)
- $1 + 1 = 10$ in binary (sum digit is 0, carry is 1)

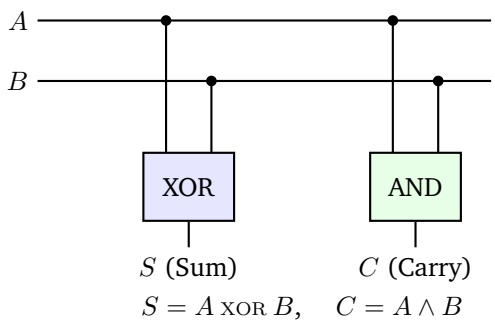
The result has two parts: a **sum bit** (S) and a **carry bit** (C).

A	B	S (Sum)	C (Carry)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Look carefully at these columns (remember, 1 means T and 0 means F):

- The Sum column matches XOR: $S = A \text{ XOR } B$.
- The Carry column matches AND: $C = A \wedge B$.

Half Adder Circuit



This circuit is called a **half adder** because it handles only half of the job. When we add longer binary numbers column by column, each column (except the rightmost) must also add the *carry* coming from the column to its right — three bits in all. A circuit that adds three bits A, B, and a carry-in C_{in}, producing a sum bit S and a carry-out C_{out}, is called a **full adder**. You'll work out how it behaves in the problems. Chaining full adders, one per column, gives a circuit that adds binary numbers of any length.

Quick Check

13. What does a half adder output when $A = 1$ and $B = 1$?
14. Why can't we use an OR gate to compute the carry bit?

Memory with Logic Gates: The SR Flip-Flop

Heads-up: this last section is the most demanding part of the handout. Pay extra attention to the feedback loop in the diagrams below. The idea that a circuit can have "memory" depends entirely on that loop: it is the first time we see a wire whose value is not determined purely by the current inputs — it also depends on what was happening a moment ago. If something feels confusing on first reading, walk through the five step-by-step diagrams slowly.

So far, our logic circuits have been *combinational*: the outputs depend only on the current inputs. But computers also need *memory*: circuits that can store information. The simplest memory circuit is the **SR flip-flop** (also called an SR latch).

Historical Note: The Invention of Electronic Memory

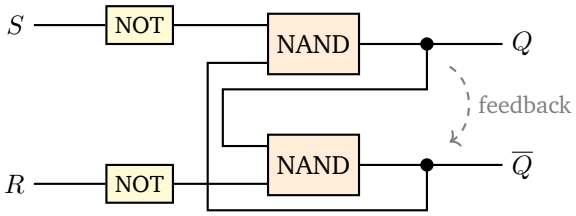
The flip-flop was invented by the British physicists **William Eccles** and **F. W. Jordan** in 1918 — decades before electronic computers existed! They called it a "trigger relay" and built it from two vacuum tubes. The name "flip-flop" came later, describing how the circuit flips between two stable states. Memory circuits like this one made electronic computers possible: without memory, a computer couldn't store intermediate results or remember instructions.

Building Memory from NOT and NAND Gates

Our SR flip-flop is built from two NAND gates, with a NOT gate on each input. This connects to our earlier result that NAND is universal — even memory circuits can be built from NANDs! The whole circuit runs on one simple fact:

If at least one input of a NAND gate is 0, its output is 1, whatever the other input is.

SR Flip-Flop (NOT-NAND Implementation)



The top NAND gate gets $\overline{S}$ (the NOT of S) and the feedback wire $\overline{Q}$; its output is Q. The bottom NAND gate gets $\overline{R}$ and the feedback wire Q; its output is $\overline{Q}$. The inputs are:

- S (Set): when $S = 1$, Q becomes 1;
- R (Reset): when $R = 1$, Q becomes 0.

The outputs are Q and $\overline{Q}$ (its complement). In normal operation, these are always opposite.

How It Works

The NOT gates invert the inputs before they reach the NAND gates. The key ingredient is the **feedback loop**: the output of each NAND gate is an input of the other. This creates *stable states* that persist even after the inputs return to 0.

- **Set** ($S = 1$, $R = 0$): $\overline{S} = 0$, so the top gate outputs $Q = 1$ no matter what. Then the bottom gate gets $\overline{R} = 1$ and $Q = 1$, so $\overline{Q} = 0$.
- **Reset** ($S = 0$, $R = 1$): $\overline{R} = 0$, so the bottom gate outputs $\overline{Q} = 1$; then the top gate gets 1 and 1, so $Q = 0$.
- **Hold** ($S = 0$, $R = 0$): both NOT gates output 1, so each NAND gate simply negates the feedback wire: $Q = \neg\overline{Q}$ and $\overline{Q} = \neg Q$. Both $Q = 1, \overline{Q} = 0$ and $Q = 0, \overline{Q} = 1$ are consistent, so the circuit keeps whichever state it was in.

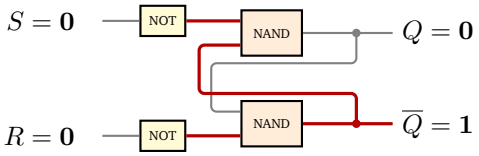
S	R	$\overline{S}$	$\overline{R}$	Q	$\overline{Q}$	Description
0	0	1	1	Q_{prev}	$\overline{Q}_{\text{prev}}$	**Hold**: memory keeps its state
1	0	0	1	1	0	**Set**: Q becomes 1
0	1	1	0	0	1	**Reset**: Q becomes 0
1	1	0	0	1	1	**Invalid**: both outputs 1 (forbidden!)

Step-by-Step Operation

Let's trace through a complete sequence showing how the flip-flop stores and remembers a bit. In the pictures, **red** wires carry 1 and **gray** wires carry 0.

Step 1: Starting state

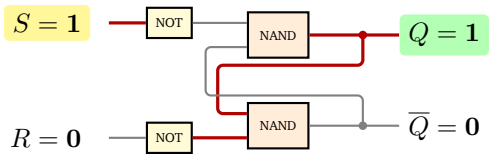
$S = 0, R = 0;\ Q = 0, \overline{Q} = 1$



Memory holds $Q = 0$

Step 2: Set pulse

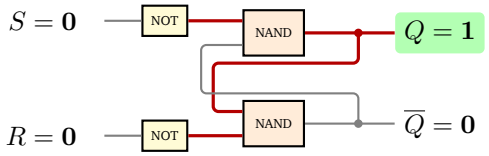
$S = \mathbf{1}, R = 0 \Rightarrow Q$ becomes **1**



$S = 1$, so $\overline{S} = 0$, so the top NAND outputs 1: $Q = 1$

Step 3: Release set

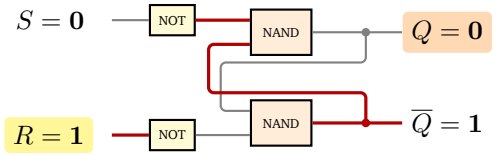
$S = 0$, $R = 0 \Rightarrow Q$ **stays 1** (memory!)



Feedback from $Q = 1$ keeps $\overline{Q} = 0$, which keeps $Q = 1$

Step 4: Reset pulse

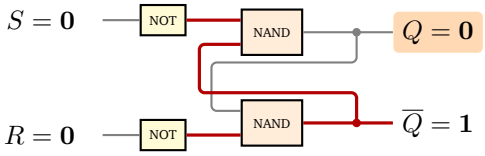
$S = 0$, $R = 1 \Rightarrow Q$ becomes **0**



$R = 1$, so $\overline{R} = 0$, so the bottom NAND outputs 1: $\overline{Q} = 1$, then $Q = 0$

Step 5: Release reset

$S = 0$, $R = 0 \Rightarrow Q$ **stays 0** (memory!)



Feedback from $\overline{Q} = 1$ keeps $Q = 0$

The key insight: when both S and R are 0, the **feedback loop** maintains whatever state was last set. This is how the circuit "remembers"!

Quick Check

15. An SR flip-flop has $Q = 1$. What inputs would reset it to $Q = 0$?
16. Why is $S = R = 1$ called the "forbidden" or "invalid" state?

Why Flip-Flops Matter

Flip-flops are the building blocks of a computer's fastest memory:

- One flip-flop stores one **bit** of information.
- 8 flip-flops store one **byte**.
- The registers and caches inside a processor hold millions of bits in flip-flop-like cells. (A computer's main memory, its RAM, uses a cheaper trick: each bit is a tiny electric charge that must be refreshed many times per second.)

More advanced flip-flops (such as D flip-flops and JK flip-flops) are built from SR flip-flops; they avoid the invalid state and add a "clock" signal so that millions of them can change in step.

Key Takeaways

- A **tautology** is always true (e.g. $A \vee \neg A$); a **contradiction** is always false (e.g. $A \wedge \neg A$).
- **XOR** ($\oplus$): true when exactly one input is true; $A \text{ XOR } B \equiv (A \vee B) \wedge \neg(A \wedge B)$. A chain of XORs is true exactly when an odd number of inputs are true.
- **NAND**: $A \text{ NAND } B \equiv \neg(A \wedge B)$. **NOR**: $A \text{ NOR } B \equiv \neg(A \vee B)$. Each of them alone is **universal**: it can build NOT, AND, OR, and therefore every logic operation.
- **XNOR**: $A \text{ XNOR } B \equiv \neg(A \text{ XOR } B)$, true when A and B have the same value; $X \equiv Y$ exactly when $X \text{ XNOR } Y$ is a tautology.
- **Logic gates** are physical circuits that implement logic operations (1 = T, 0 = F).
- A **half adder** uses XOR for the sum and AND for the carry; a **full adder** also adds a carry-in.
- An **SR flip-flop** is a memory circuit: Set ($S = 1$) stores 1, Reset ($R = 1$) stores 0, and the state persists when both inputs are 0.

Common Mistakes

- **Confusing OR and XOR**: OR is true when at least one input is true; XOR is true when exactly one is true.
- **Forgetting NAND/NOR truth values**: NAND is false only when both inputs are true; NOR is true only when both inputs are false.
- **Confusing XNOR with NOR**: "neither A nor B" is NOR. XNOR means "A and B have the same value."
- **Calling a formula a tautology after checking a few rows**: a tautology must be true in *every* row. (One false row is enough to show it is *not* a tautology.)

Practice Problems

1. Determine whether each is a tautology, a contradiction, or neither:
   (a) $(A \wedge B) \vee (\neg A \vee \neg B)$
   (b) $(A \vee B) \wedge \neg A \wedge \neg B$
   (c) $(A \wedge \neg B) \vee B$
2. Answer each question and give an example.
   (a) If X is a tautology, what can you say about $\neg X$?
   (b) Find two formulas, neither of them a contradiction, whose AND is a contradiction.
   (c) Find two formulas, neither of them a tautology, whose OR is a tautology.
3. Write the complete truth table for XOR, including intermediate columns, showing that it equals $(A \wedge \neg B) \vee (\neg A \wedge B)$.
4. Evaluate each expression:
   (a) $(T \text{ XOR } T) \text{ XOR } F$
   (b) $T \text{ NAND } (F \text{ NAND } T)$
   (c) $(T \text{ NOR } F) \text{ NOR } T$
5. Show that $A \text{ XOR } B \equiv (A \vee B) \wedge (A \text{ NAND } B)$ using a truth table.
6. **The staircase light.** A staircase light has one switch at the bottom of the stairs and one at the top. Let A be "the bottom switch is up" and B be "the top switch is up." Flipping either switch (either way) changes the light, and when both switches are down, the light is off.
   (a) Make a table of $L =$ "the light is on" for all four positions of the switches.
   (b) Which operation is L?
   (c) A different electrician wires the light so that it is *on* when both switches are down. Now which operation is L?
7. For each operation XOR, NAND, NOR, XNOR, simplify A op A (the operation applied to A and itself).
8. (a) Compute $(T \text{ NAND } T) \text{ NAND } F$ and $T \text{ NAND } (T \text{ NAND } F)$.
   (b) Is NAND associative? Is XOR associative?
9. Using only NAND gates, build $A \text{ NOR } B$. How many gates does your circuit use?
10. Using only NOR, build $A \wedge \neg B$ with just two gates. (*Hint:* De Morgan.)
11. A circuit has three inputs A, B, C. The first NAND gate takes A and B. The second NAND gate takes the output of the first gate and C; its output is the output of the circuit.
    (a) Make the truth table of the circuit.
    (b) Show that the output is equivalent to $(A \wedge B) \vee \neg C$.
12. **A secret code.** Alice and Bob share the secret key `0110`. To send a 4-bit message, Alice XORs each message bit with the matching key bit (1 means T, 0 means F).
    (a) Encode the message `1011`.
    (b) Bob XORs the coded message with the same key, bit by bit. What does he get?
13. **Parity bits.** To catch transmission errors, computers often add a *parity bit* at the end of a message: an extra bit chosen so that the total number of 1s is even.

(a) Find the parity bit for the messages `1011001` and `1110000`.

(b) Each 8-bit word `10110111` and `11010110` was sent with its parity bit as the last bit. Which one surely contains an error?

(c) Explain why the parity bit is the XOR of all the message bits.

14. Show that $(A \text{ XOR } B) \vee (A \text{ XNOR } B)$ is a tautology and $(A \text{ XOR } B) \wedge (A \text{ XNOR } B)$ is a contradiction. Can you explain it without a truth table?

15. Show that $A \text{ XNOR } \neg B \equiv A \text{ XOR } B$. Explain in words why this makes sense.

16. **Comparing bits.** A and B are 1-bit numbers (1 = T, 0 = F). Write formulas for G: "$A > B$," E: "$A = B$," and L: "$A < B$." Check that in every row exactly one of G, E, L is true.

17. **Museum alarm.** A museum alarm rings if the motion sensor detects movement (M) while the system is armed (R), or if the glass sensor detects broken glass (G), armed or not.

    (a) Write a formula for "the alarm rings" using AND and OR.

    (b) Build the same circuit from three NAND gates only, and check it.

18. Add the binary numbers `1011` and `0110` column by column, starting from the right. For each column, list the three bits being added (including the carry-in) and the resulting sum bit and carry-out. Check your answer in base ten.

19. **Full adder.** A full adder adds three bits: A, B, and a carry-in C_{in} from the previous column. It produces a sum bit S and a carry-out C_{out}; for example, $1 + 1 + 1 = 11$ in binary, so $S = 1$ and $C_{\text{out}} = 1$.

    (a) Complete the 8-row truth table for a full adder.

    (b) Express S in terms of A, B, C_{in} using XOR.

    (c) In words, when is $C_{\text{out}} = 1$?

20. An SR flip-flop (built as in this handout) starts with $Q = 1$. The inputs $(S, R) = (0, 0)$, $(0, 1)$, $(1, 0)$, $(0, 0)$, $(0, 1)$, $(0, 0)$ are applied, in this order. What is Q after each step?

Practice Problems: Solutions

1. (a) Tautology: $(A \wedge B) \vee (\neg A \vee \neg B) \equiv (A \wedge B) \vee \neg(A \wedge B)$ by De Morgan, which has the form $X \vee \neg X$.
   (b) Contradiction: we can't have both "A or B is true" and "both A and B are false."
   (c) Neither: it is equivalent to $A \vee B$ (if B is true, the formula is true; if B is false, it is just A). It is false when $A = B = F$ and true when $A = T$.
2. (a) $\neg X$ is a contradiction: every T in the column of X becomes F. Example: $\neg(A \vee \neg A)$.
   (b) A and $\neg A$: neither is a contradiction, but $A \wedge \neg A$ is.
   (c) A and $\neg A$ again: neither is a tautology, but $A \vee \neg A$ is.
3. XOR truth table with intermediate steps:

A	B	$\neg A$	$\neg B$	$A \wedge \neg B$	$\neg A \wedge B$	$(A \wedge \neg B) \vee (\neg A \wedge B)$	A XOR B
T	T	F	F	F	F	F	F
T	F	F	T	T	F	T	T
F	T	T	F	F	T	T	T
F	F	T	T	F	F	F	F

   The last two columns match.
4. (a) $(T \text{ XOR } T) \text{ XOR } F = F \text{ XOR } F = F$
   (b) $T \text{ NAND } (F \text{ NAND } T) = T \text{ NAND } T = F$
   (c) $(T \text{ NOR } F) \text{ NOR } T = F \text{ NOR } T = F$
5. Truth table:

A	B	$A \vee B$	A NAND B	$(A \vee B) \wedge (A \text{ NAND } B)$
T	T	T	F	F
T	F	T	T	T
F	T	T	T	T
F	F	F	T	F

   The last column F, T, T, F matches the XOR truth table. (This is just our first XOR formula, with $\neg(A \wedge B)$ written as A NAND B.)
6. (a) Both down (F, F): off. From there, flipping one switch turns the light on, so (T, F) and (F, T): on. Flipping the other switch too changes it again, so (T, T): off.
   (b) $L = A$ XOR B (on exactly when the switches are in different positions).
   (c) $L = A$ XNOR B (on exactly when the switches are in the same position).
7. $A \text{ XOR } A \equiv F$; $A \text{ NAND } A \equiv \neg A$; $A \text{ NOR } A \equiv \neg A$; $A \text{ XNOR } A \equiv T$.
8. (a) $(T \text{ NAND } T) \text{ NAND } F = F \text{ NAND } F = T$, but $T \text{ NAND } (T \text{ NAND } F) = T \text{ NAND } T = F$.
   (b) NAND is *not* associative: part (a) shows that moving the parentheses can change the answer. XOR *is* associative (it is in the box of XOR properties; both sides are true exactly when an odd number of A, B, C are true).
9. Since $A \text{ NOR } B = \neg(A \vee B)$, build OR and then NOT it. OR takes 3 NAND gates:

$$X = (A \text{ NAND } A) \text{ NAND } (B \text{ NAND } B) \equiv A \vee B.$$

   Then $A \text{ NOR } B \equiv X \text{ NAND } X$. Total: **4 gates**.
10. $(A \text{ NOR } A) \text{ NOR } B$. Check: $A \text{ NOR } A = \neg A$, and $\neg A \text{ NOR } B = \neg(\neg A \vee B) \equiv \neg\neg A \wedge \neg B \equiv A \wedge \neg B$ by De Morgan.

11. (a) Rows ABC = TTT, TTF, TFT, TFF, FTT, FTF, FFT, FFF: first gate $A \text{ NAND } B$ = F, F, T, T, T, T, T, T; output = **T, T, F, T, F, T, F, T**.

    (b) Output $= \neg\big(\neg(A \wedge B) \wedge C\big) \equiv \neg\neg(A \wedge B) \vee \neg C \equiv (A \wedge B) \vee \neg C$ (De Morgan, then double negation).

12. (a) `1011` XOR `0110` = `1101`.

    (b) `1101` XOR `0110` = `1011`: the original message! (XORing with the same key twice changes nothing, since $K \text{ XOR } K \equiv F$.)

13. (a) `1011001` has four 1s, so its parity bit is **0**. `1110000` has three 1s, so its parity bit is **1**.

    (b) `10110111` has six 1s (even): no error detected. `11010110` has five 1s (odd): it surely contains an error.

    (c) The parity bit must be 1 exactly when the message has an odd number of 1s. A chain of XORs is 1 (true) exactly when an odd number of its inputs are 1. So the parity bit equals the XOR of all the message bits.

14. Since $A \text{ XNOR } B \equiv \neg(A \text{ XOR } B)$, the formulas have the form $X \vee \neg X$ (always true) and $X \wedge \neg X$ (always false), with $X = A \text{ XOR } B$. A truth table confirms it: the columns of $A \text{ XOR } B$ and $A \text{ XNOR } B$ are F, T, T, F and T, F, F, T, so their OR is all T and their AND is all F.

15. Truth table (rows TT, TF, FT, FF): $\neg B$ = F, T, F, T; $A \text{ XNOR } \neg B$ = F, T, T, F, the same as $A \text{ XOR } B$. In words: "A has the same value as $\neg B$" means "A has the opposite value of B," which is "A and B are different."

16. $G = A \wedge \neg B$ (only the row $A = 1$, $B = 0$); $E = A \text{ XNOR } B$; $L = \neg A \wedge B$. Rows (A, B) = (1,1): only E; (1,0): only G; (0,1): only L; (0,0): only E. In every row exactly one of them is true.

17. (a) Alarm $= (M \wedge R) \vee G$.

    (b) $(M \text{ NAND } R) \text{ NAND } (G \text{ NAND } G)$. Check: $G \text{ NAND } G = \neg G$, so the output is $\neg\big(\neg(M \wedge R) \wedge \neg G\big) \equiv (M \wedge R) \vee G$ by De Morgan. (This trick — replacing an "AND then OR" circuit by NAND gates — works in general.)

18. Columns from right to left:

column	A	B	carry-in	sum bit	carry-out
1st (rightmost)	1	0	0	1	0
2nd	1	1	0	0	1
3rd	0	1	1	0	1
4th	1	0	1	0	1

    The last carry-out becomes a new leading digit: `1011 + 0110 = 10001`. Check: $11 + 6 = 17 = 16 + 1$. ✓

19. (a) Truth table:

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

    (b) $S = A \text{ XOR } B \text{ XOR } C_{\text{in}}$: the sum bit is 1 exactly when an odd number of the inputs are 1.

    (c) $C_{\text{out}} = 1$ exactly when at least two of the three inputs are 1. (It is the "majority vote" of the three bits.)

20. Q = **1, 0, 1, 1, 0, 0**: hold (1), reset (0), set (1), hold (1), reset (0), hold (0).

Homework

1. Determine whether each is a tautology, a contradiction, or neither:

   (a) $(A \text{ NAND } B) \vee A$

   (b) $(A \text{ NOR } B) \wedge A$

   (c) $(A \text{ XOR } B) \text{ XOR } (A \text{ XNOR } B)$

   (d) $(A \text{ NAND } B) \vee (A \text{ NOR } B)$

2. Simplify each expression to T, F, A, or $\neg A$:

   (a) $A \text{ NAND } T$

   (b) $A \text{ NOR } F$

   (c) $A \text{ XOR } \neg A$

   (d) $A \text{ NAND } \neg A$

   (e) $A \text{ NOR } \neg A$

   (f) $(A \text{ XOR } B) \text{ XOR } B$

3. (a) Show with a truth table that $A \text{ XOR } B \equiv (A \text{ NAND } N) \text{ NAND } (B \text{ NAND } N)$, where $N = A \text{ NAND } B$. Draw the circuit using the gate symbols from this handout. How many NAND gates does it use?

   (b) Suppose instead you translated $(A \wedge \neg B) \vee (\neg A \wedge B)$ into NAND gates piece by piece (NOT = 1 gate, AND = 2 gates, OR = 3 gates). How many gates would that take?

4. M Using only NOR gates, build a circuit for $A \text{ XOR } B$. (It can be done with 5 gates.)

5. **Mystery circuit.** An XOR gate and an AND gate both receive the inputs A and B. Their two outputs go into a second XOR gate, whose output is the output of the circuit. Which basic operation does this circuit compute? Explain why.

6. **Three switches.** A long hallway has three doors, each with a switch for the same ceiling light. Flipping any one switch changes the light, and when all three switches are down, the light is off. Let A, B, C mean "switch 1 / 2 / 3 is up."

   (a) Explain why the light is on exactly when an odd number of switches are up.

   (b) Write a formula for "the light is on" using XOR.

   (c) What would the formula be with four switches?

7. **Secret codes.** As in a "one-time pad," a message M is coded by XORing it, bit by bit, with a secret key K: the coded message is $C = M \text{ XOR } K$.

   (a) Code the message `110010` with the key `101011`, then decode the result with the same key.

   (b) Prove, using the properties of XOR, that $(M \text{ XOR } K) \text{ XOR } K \equiv M$ for every M and K, so decoding always works.

   (c) A spy learns both a message M and its coded version C. Show that she can now find the key.

8. **Finding an error.** Nine data bits are written in a 3×3 square. Then a parity bit is added at the end of each row, a parity bit at the bottom of each column, and one more in the corner, so that *every row and every column* of the 4×4 grid has an even number of 1s. During transmission exactly one bit was flipped, and the grid below arrived. Which bit was flipped? Explain how the parity bits find it.

1	0	1	0
1	1	1	0
1	1	1	1
0	0	1	1

9. M **Full adder from half adders.** A *half adder* takes two bits and outputs their sum bit (XOR) and carry (AND). A *full adder* adds three bits A, B, C_{in}, and outputs a sum bit S and a carry-out C_{out} so that $A + B + C_{\text{in}} = 2 \cdot C_{\text{out}} + S$ (for example, $1 + 1 + 1 = 11$ in binary).

   (a) Show how to build a full adder from two half adders and one OR gate. Describe the wiring and check all 8 rows with a table.

   (b) Explain why the OR gate could be replaced by an XOR gate without changing anything.

10. H An SR flip-flop (built as in this handout) starts with $Q = 0$. The following sequence of inputs is applied: $(S, R) = (1, 0)$, then $(0, 0)$, then $(0, 1)$, then $(0, 0)$, then $(1, 0)$. What is the value of Q after each step?

11. M **Mixed operators.**

   (a) Rewrite $(A \text{ NAND } B) \text{ NOR } C$ using only $\neg$, $\wedge$, $\vee$, and simplify as much as you can.

   (b) Do the same for $A \text{ NAND } (B \text{ NOR } C)$.

   (c) Describe in words exactly when $(A \text{ XOR } B) \text{ XNOR } C$ is true.

   (d) Show that $(A \text{ XNOR } B) \text{ XOR } C \equiv (A \text{ XOR } B) \text{ XNOR } C$.

12. H **A Knight-or-Knave detector.** On the island of knights (who always tell the truth) and knaves (who always lie), you may ask an islander yes/no questions. Write "yes" as T and "no" as F, and let N be the statement "this islander is a knave."

   (a) Explain why, if you ask "Is P true?", the answer you hear is $P \text{ XOR } N$.

   (b) You are at a fork in the road and want to know whether the left road leads to the village (call this statement V), but you don't know whether the islander is a knight or a knave. Show that if you ask "Is $V \text{ XOR } N$ true?" — in words, "Is exactly one of these true: the left road leads to the village, you are a knave?" — the answer you hear is always V.

Quick Check Answers

1. Yes. $A \vee \neg A$ is already always true, and $T \vee B \equiv T$.
2. A contradiction: $A \vee \neg A$ is always true, so its negation is always false.
3. No (it is neither). For $A = T$, $B = F$: $(T \wedge F) \vee \neg T = F \vee F = F$.
4. $(T \text{ XOR } F) \text{ XOR } T = T \text{ XOR } T = F$
5. $T \text{ XOR } T = F$ (both true, so XOR is false)
6. Three of the five inputs are T, an odd number, so the answer is T.
7. $T \text{ NAND } F = \neg(T \wedge F) = \neg F = T$
8. $T \text{ NOR } T = \neg(T \vee T) = \neg T = F$
9. $A \vee B \equiv (A \text{ NOR } B) \text{ NOR } (A \text{ NOR } B)$. Since $A \text{ NOR } B = \neg(A \vee B)$ and $X \text{ NOR } X = \neg X$, we get $\neg(\neg(A \vee B)) \equiv A \vee B$.
10. First $T \text{ XNOR } F = \neg(T \text{ XOR } F) = \neg T = F$. Then $F \text{ XNOR } T = \neg(F \text{ XOR } T) = \neg T = F$. So the answer is F.
11. Exactly when $B = F$, for either value of A (by the property $A \text{ XNOR } F \equiv \neg A$; if $B = T$, then $A \text{ XNOR } B \equiv A$, which is never equal to $\neg A$).
12. By De Morgan's law, $\neg(A \wedge B) \equiv \neg A \vee \neg B$: the two sides agree in every row, so their XNOR is T in every row.
13. $S = 0$, $C = 1$ (because $1 + 1 = 10$ in binary).
14. OR gives 1 when $A = 1$, $B = 0$, but there is no carry then. The carry is 1 only when *both* bits are 1, which is AND.
15. Set $S = 0$ and $R = 1$ (then return to $S = R = 0$; the flip-flop remembers $Q = 0$).
16. When $S = R = 1$, both NOT gates output 0, so both NAND gates receive a 0 input and both outputs become 1. Then $Q = 1$ and $\overline{Q} = 1$, which breaks the rule that Q and $\overline{Q}$ are opposites. Worse, if both inputs then return to 0 at the same moment, the final state is unpredictable — it depends on which gate reacts faster.