

REDSTONE
UNIVERSITY

Welcome to Redstone University!

Have you ever used a computer or a smartphone and wondered what's *really* happening inside? Not just the software, but the deep, physical magic of a machine that seems to "think"?

This isn't just another Minecraft course. This is a journey into the heart of the machine.

As a non-traditional, self-taught software engineer, I found myself wanting to explore the foundational principles of computer science. I realized that the abstract concepts of binary, logic gates, and computer architecture were difficult to grasp from books and theory alone. At the same time, I saw the incredibly complex and logical machines being built in Minecraft with Redstone. The idea was born: **what if we could learn how a computer works by building one from scratch, using tools we already love?**

That is the mission of Redstone University. We will make the abstract tangible. We will turn theory into a physical, working machine that you can walk around inside of.

My Personal Journey & Course Philosophy

Redstone University is the product of my own adventure learning digital logic and computer architecture. This adventure started with curiosity and grew into a passion for building, experimenting, and teaching. Every lesson, every build, and every design choice in this course is shaped by what felt intuitive and exciting to me as a learner. I've structured the curriculum to follow the path that made the most sense to me: building what I wanted to see next, solving the problems that naturally arose, and always striving to make each concept click in a hands-on, visual way.

What sets this course apart?

- It's grounded in *real experience*: you'll follow the same journey I did, learning not just the "what" but the "why" and "how" behind each step.
 - We use **Minecraft** as our laboratory, making abstract concepts tangible and fun.
 - We focus on clarity and intuition, not just efficiency or speed.
-

Course Build Philosophy

Disclaimer: The builds and circuits in this course are intentionally designed for clarity and educational value, not for performance or compactness. We lay out circuits horizontally and in a "paper-like" fashion to make the logic easy to follow, just as you would draw them on paper. Our goal is to illustrate the underlying principles of computer engineering, not to create the most efficient or smallest circuits.

How the Course is Structured

This course is organized as a complete curriculum, taking you from zero knowledge to a fully functional, programmable 4-bit computer. It is divided into Parts (major phases), Modules (specific projects), and Lessons (step-by-step instructions). Each module builds a piece of our computer, and each lesson guides you through that process.

You'll find:

- **Personal motivation and narrative:** Each module is introduced with a story or challenge that mirrors my own learning process.
- **Hands-on builds:** Every concept is brought to life with a Minecraft circuit and, where helpful, a CircuitVerse diagram.
- **Theory and practice:** The modules balance foundational theory with immediate, practical application.

- **Real-world and software connections:** You'll see how each idea relates to real computers and even to programming challenges (like those on LeetCode).
-

The Journey Ahead

- **Part I: The Foundations - Speaking to the Machine.** We will begin by building the essential human-computer interface. We'll learn the language of binary, the grammar of Boolean logic, and construct our own "keyboard" and "monitor."
- **Part II: Engineering a Robust Arithmetic Unit.** Here, we will build the mathematical core of our machine. We'll engineer an adder and subtractor, discover our machine's natural limitations through "bugs" like overflow, and upgrade our system to solve them, just like real engineers.
- **Part III: The Processor Core.** With our arithmetic unit perfected, we will forge the true brain of our computer: the Arithmetic Logic Unit (ALU). We will combine all our mathematical and logical circuits into one powerful, versatile, and controllable component.
- **Part IV: Creating an Automated Computer.** In the final core modules, we'll give our processor a memory to store its thoughts and a clock to act as its heartbeat. We will assemble everything into a single, automated machine that can run a simple program on its own.
- **Part V: Post-Graduate Studies.** For those who want to go even further, we'll explore advanced topics, tackling the complex challenge of making our computer display multi-digit decimal numbers, just like a real-world calculator.

Who Is This For?

This course is for the curious. It's for:

- **My daughter, Ada,** for whom this project was first imagined.
- **Students and kids** who want a fun, hands-on introduction to STEM and computer science.
- **University CS students** who want a physical way to visualize the concepts from their "Computer Architecture" class.
- **Self-taught programmers and professionals** who want to solidify their understanding of what's happening at the hardware level.

How to Get Started & Accessibility

This course is designed to be followed along in **Minecraft**. However, Minecraft is not strictly required!

For each module, I will provide guidance, and I also provide a **World Download** (the "RU Campus") with the completed circuits. You can use this to check your work, explore the final product, or use the pre-built components as "black boxes" if you want to focus more on the high-level concepts.

The "No-Minecraft Track": If you don't have Minecraft or prefer a more theoretical approach, you can still complete this entire course. Every lesson will include text descriptions, diagrams, and schematics. I will also provide links to free online digital logic simulators (like [CircuitVerse](#)) where you can build and test these circuits without the game. The core learning is in the logic, not just the blocks.

I am excited for you to join me on this journey. It's time to stop just *using* computers and start *understanding* them.

How to Use This Course

- **Follow the modules in order:** Each module builds on the last, so start at the beginning and work your way through.
- **Try the builds yourself:** The hands-on experience is where the real learning happens. Use Minecraft or CircuitVerse as you prefer.
- **Use the world download or diagrams:** If you get stuck or want to check your work, explore the provided world or reference the diagrams.
- **Read the real-world and software connections:** These sections help you see why each concept matters beyond Minecraft.
- **Go at your own pace:** Take your time with each lesson, and revisit earlier modules whenever you need a refresher.

Ready? Let's get building!

Part I: The Foundations, Speaking to the Machine

Welcome to Part I of Redstone University's epic journey to build a working computer from scratch! Our grand ambition is to create a fully functional machine, but every masterpiece starts with a strong foundation. In this part, we're diving into the Human-Computer Interface, the critical components that let us (as humans) communicate with our digital creation.

By the end of Part I, our computer won't be thinking on its own yet, but it will have a complete input and output system. You'll be able to send it numbers and see those numbers displayed in a way that's instantly clear to you. This is where the magic begins!

Our Mission for Part I

We'll conquer this foundation in three exciting modules, blending hands-on building with powerful theory and culminating in a show-stopping application:

- **Module 1: The Input Register** Build the computer's keyboard, a simple set of levers to input numbers in binary (the machine's native language).
- **Module 2: Boolean Algebra** Take a crucial dive into the theory that powers all digital logic. This is the course's most important lecture, where you'll master the grammar of NOT, AND, OR, and XOR, the rules behind every circuit you'll design.
- **Module 3: Decoders and Displays** Apply your new theoretical skills to a major challenge: creating a two-stage translator that converts binary into human-readable numbers on a stunning 7-segment display.

This part is crafted to deliver a thrilling payoff. You'll start with basic switches and end with a device that feels alive. These concepts are the bedrock (pun intended) for everything to come.

Why This Progression?

I've designed this course to spark your motivation early. Personally, I wanted to see my inputs and outputs light up on a 7-segment display to confirm my work was correct. It made the abstract ideas of binary and logic feel real and rewarding. That's why we begin with the input register and quickly move to the display. It's a tangible goal that keeps you hooked.

This approach mirrors a core belief about learning to code or build: the faster you see something working, the more driven you'll be to push forward. Part I is all about giving you that instant sense of progress and accomplishment.

Ready to start? Let's build our first component: the Input Register!

Module 1: Speaking in 1s and 0s – The Input Interface

Module Summary

- **Narrative Beat:** Before we can build a computer, we need a way to talk to it. Our language will be binary, and our input interface will be a set of simple levers.
 - **Learning Goals:**
 - Understand binary as a system of on/off switches.
 - Build a physical interface to input binary numbers.
 - Strengthen binary intuition through practice.
 - **Lesson Overview:**
 - Lesson 1.1: The Theory – Why Computers Use Binary
 - Lesson 1.2: The Lab – Building and Using Our 4-Bit Input Interface
 - Lesson 1.3: Drills & Games – Strengthening Your Binary Intuition
 - Lesson 1.4: Module 1 Checkpoint
 - **Minecraft Artifact:** A working 4-bit input interface for binary numbers.
-

Module Introduction

Welcome to your first day at Redstone University!

Our grand adventure is to build a complete, working computer from scratch. But like any great journey, we need to start with the basics. The very first thing we need is a way to talk to our machine. We need a way to give it information.

In this module, we're going to build a **4-bit input interface**, a simple set of switches that lets us speak the computer's native language: **binary**. In Minecraft, levers hold their state, making them perfect for this job. By flipping them, we can set a 4-bit binary number (any value from 0 to 15) and see it in action. This isn't a true register (a storage device we'll build later), but it's a hands-on way to input binary data and understand how computers start processing information. As we move forward, you'll see how this simple setup connects to the bigger picture.

Let's get started!

Lesson 1.1: The Theory – Why Computers Use Binary

Think about how you count. You probably use ten symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. This is the **decimal** (base-10) system. It feels natural to us, likely because humans evolved with ten fingers. When we get past 9, we don't invent a new symbol; we just add a new column to the left, the "tens" column, and start over. The number 12 is really just our way of saying "one ten, plus two ones."

Computers are different. They don't have fingers. Deep down, they are made of billions of microscopic electronic switches called transistors. A switch is a very simple device. It can only ever be in one of two states: **ON** or **OFF**. There is no "halfway on."

This simple, two-state system is the foundation of all modern computing. We call it **binary** (base-2). To represent any piece of information, we just assign a meaning to these two states:

- OFF = 0
- ON = 1

That's it! Every single thing your computer does, from displaying this text, to playing a song, to running a complex game, is ultimately just a massive, coordinated manipulation of these simple 1s and 0s. Each individual 1 or 0 is called a **bit** (short for "binary digit").

So, how can we possibly represent a big number like 13 with just 1s and 0s? We use the same trick as our decimal system: we use columns with different values. But instead of ones, tens, and hundreds, our binary columns simply double each time.

Bit Position	3	2	1	0
--------------	---	---	---	---

Bit Position	3	2	1	0
Power of 2	2^3	2^2	2^1	2^0
Place Value	8	4	2	1
Binary	1	1	0	1

- **Bit Position:** The rightmost bit is position 0, then 1, 2, and so on to the left.
- **Power of 2:** Each position represents a power of two.
- **Place Value:** The actual value for each bit.
- **Binary:** The value of each bit for the number `1101`.

To figure out the value of a binary number, you just add up the values of the columns where there is a `1` (or an "ON" switch).

For example, the binary number `1101`:

- Is there a `1` in the `8`'s place? **Yes.**
- Is there a `1` in the `4`'s place? **Yes.**
- Is there a `1` in the `2`'s place? **No.**
- Is there a `1` in the `1`'s place? **Yes.**

So, the value is `8 + 4 + 1 = 13`. We've just translated from the computer's language back to ours!

Lesson 1.2: The Lab – Building and Using Our 4-Bit Input Interface

It's time to stop talking and start building! Our **4-bit input interface** will act as a simple "keyboard," letting us manually input any number from 0 to 15 in binary. Using levers, we will set the bits by flipping them up for `1` and down for `0`. A simple setup that will enable us to create binary numbers we can see and use.

Materials Needed

- 4 standard building blocks*
- 4 Levers
- 4 Signs
- A few pieces of Redstone Dust

*You can use any solid block, but for the input interface, I recommend a redstone lamp. It doubles as a visual indicator of the current state of each bit.

The Build Guide



Figure: The input interface in Minecraft, set to `0110` (binary for 6). The levers are flipped to represent the bits, and the dust is connected to the back. Using redstone lamps makes it easy to see the current state of each bit.

1. I recommend creating a new world and under the advanced options, set the world type to "Flat". They even have a flat preset called "Redstone Ready" that is perfect for our needs.
2. Place **four Redstone Lamps** or **four solid blocks** in a horizontal line with one space between to prevent their redstone dust from merging.
3. On the front face of each block, place one **Lever**. A lever is the perfect physical bit! When it's flipped down, it's `0`. When it's flipped up, it's `1`.
4. Now, let's label our work so we don't get confused. Place a **Sign** on the very top of the block. From **right to left**, label them `1`, `2`, `4`, and `8`. We go right-to-left because, just like in the number `12`, the least valuable digit (the `2`) is on the right. See the schematic, screenshot, or diagram for clarity if needed.
5. Finally, let's wire it up. Go around to the back of your four blocks to the opposite side that you placed the lever. Place a piece or two of **Redstone Dust** on the ground directly behind each one. When you flip a lever, its block becomes powered, which sends a signal to the dust. These four parallel lines of dust are now your official **4-bit input bus**. A "bus" is just the fancy engineering term for a bundle of wires that carry a complete piece of information.
6. Double-check that your build looks similar to the one in the figure above.

Before we test our new input interface, I want to introduce you to the same input interface represented in CircuitVerse, a free online digital logic circuit simulator. Moving forward, every circuit we build will be introduced in theory with the circuitverse version first, and then we will build it in Minecraft. This is primarily due to being able to easily represent the circuit in a clear and concise way, something that isn't always possible with Minecraft screenshots. Everything you build is included in the [circuitverse project for this course](#).

CircuitVerse Version

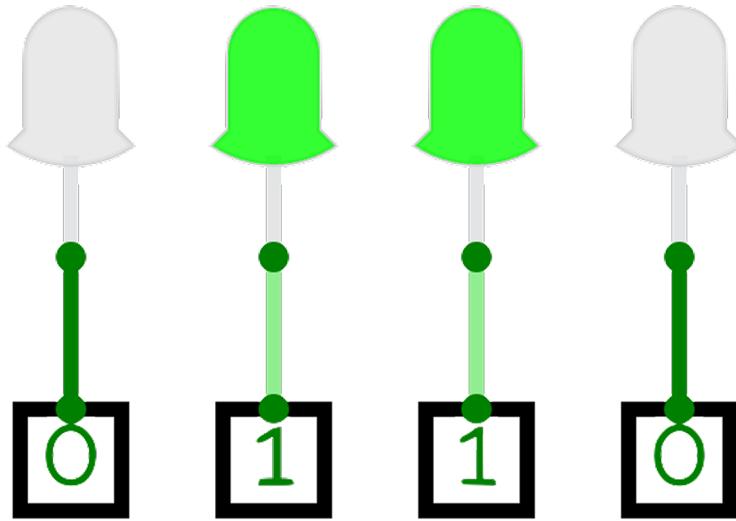


Figure: The same 4-bit input interface, built in CircuitVerse. It is also set to `0110` (6 in decimal).

While it has a few stylistic differences, the concept is exactly the same as our Minecraft build. It's an input interface that allows for input of a 4-bit binary number.

Don't worry, we will be building more interesting circuits very soon.

Lesson 1.3: Drills & Games – Strengthening Your Binary Intuition

Let's get a feel for our new device. Binary feels weird at first, but it will become second nature with just a little practice.

Takeaway: Practicing will make binary numbers feel as natural as decimal. The more you practice, the faster you'll get!

Drill 1: Binary to Decimal

- **Goal:** What decimal number is `1011` ?
- **Action:** Go to your input interface and set the levers: `ON, OFF, ON, ON` .
- **Calculation:** $8 + 0 + 2 + 1 = 11$. So, `1011` is `11` .

Drill 2: Decimal to Binary (The "Greedy" Method)

- **Goal:** Let's represent the number `6` .
- **Thought Process:** Always start with your biggest bit and work your way down.
 - i. Is `6` greater than or equal to `8` ? **No.** Leave the `8` lever OFF.
 - ii. Is `6` greater than or equal to `4` ? **Yes.** Flip the `4` lever ON. We have $6 - 4 = 2$ left to account for.
 - iii. Is `2` greater than or equal to `2` ? **Yes.** Flip the `2` lever ON. We have $2 - 2 = 0$ left.
 - iv. Is `0` greater than or equal to `1` ? **No.** Leave the `1` lever OFF.
- **Result:** The levers are `OFF, ON, ON, OFF` , which is the binary number `0110` .

The Binary "Game"

While not the ideal version of a game, this is a great way to build speed. Pick a random number between `0` and `15` and see how quickly you can represent it on your input interface. This will burn the powers of two (`1`, `2`, `4`, `8`) into your memory.

Lesson 1.4: Module 1 Checkpoint

Let's check our understanding before moving on.

Takeaway: If you can answer these questions, you're ready to move on to the next big idea: logic!

Quiz

1. What is the largest number a `5`-bit input interface could input? (Hint: The next bit would be the `16`s place).
2. What is the decimal value of the binary number `1100`?
3. How would you represent the number `10` in binary?

Real-World Connection: CPU Registers

Your **4-bit input interface** is a simplified version of how real computers get information from the world. In everyday life, devices like keyboards, mice, and sensors act as input interfaces, turning your actions (like typing or clicking) into binary signals the computer understands. Our Minecraft build uses four levers to input a 4-bit number (0 to 15), but imagine scaling that up. Modern computers often handle **64-bit data**, meaning their circuits can process 64 bits at once, enough to represent numbers bigger than 18 quintillion!

Here's how it connects: once an input device sends binary data, the computer stores it in **registers**, tiny, super-fast storage units inside the CPU. A "64-bit processor" has registers that hold 64 bits, letting it crunch huge numbers or instructions in a single step. Your 4-bit interface is just the beginning, it's how we "talk" to the machine. Later, we'll build a register and see how they use that input to make the computer think!

Software Connection (LeetCode): Counting Bits

How does a programmer "look at" the individual bits you just set with your levers? They use bitwise operations! This is a sneak peek of what we'll learn in Module 2, but it's too cool not to share.

A classic LeetCode problem is "**Number of 1 Bits**": count how many `1`s are in a number's binary representation. Programmers solve this by checking each bit of the number one by one. It also gives a sneak peek at the concept of bitwise operations, which are essential for low-level programming and optimization.

```
def countSetBits(n):  
    count = 0  
    while n > 0:  
        # The '& 1' checks if the last bit is a 1  
        if (n & 1) == 1:  
            count += 1  
        # The '>>= 1' shifts all bits one place to the right  
        n >>= 1  
    return count  
  
# The binary for 13 is 1101  
print(countSetBits(13)) # Output: 3
```

Software Analogy: In most programming languages, you can use bitwise operators to manipulate numbers at the binary level. For example, in Python, `n & 1` checks the lowest bit, and `n >>= 1` shifts all bits to the right. This is just like flipping levers and reading wires from your input interface!

Module 1 Conclusion

Fantastic work! You've now mastered the most fundamental concept in all of computing: how information is physically represented in a binary system. You have a working input device, and you've seen how this physical concept directly connects to both real-world hardware and clever software algorithms.

What's next: Right now, these are just dumb switches connected to wires. In the next module, we will learn the rules of logic that will allow us to start manipulating these signals to perform calculations and make decisions.

We will build the first real circuits that can process our binary inputs and produce outputs based on logical rules. The basic building blocks of our computer are about to take shape. Get ready for the world of logic gates and circuits!

Module 2: The Language of Logic – A Deep Dive into Boolean Algebra

Module Summary

- **Narrative Beat:** We've built our keyboard, but to make the computer *think*, we need to learn its grammar. This isn't a Minecraft lesson; this is the fundamental language of all digital electronics. Welcome to Boolean Algebra.
 - **Learning Goals:**
 - Move beyond physical blocks to understand the formal, abstract language that governs all digital circuits.
 - Understand *why* circuits work the way they do, and how to design and simplify them on paper before ever placing a block.
 - **Lesson Overview:**
 - Lesson 2.1: The Rules of Thought
 - Lesson 2.2: The Core Operators (The Verbs of Logic)
 - Lesson 2.3: The Laws of Logic & The Power of Simplification
 - Lesson 2.4: The Special Operator – XOR
 - Lesson 2.5: Software Superpowers – The XOR Trick for Programmers
 - Lesson 2.6: The Negated Gates – NAND, NOR, and XNOR
 - Lesson 2.7: Module 2 Checkpoint
 - **Minecraft Artifact:** A set of working Redstone logic gates (NOT, AND, OR, XOR, NAND, NOR, XNOR).
-

Module Introduction

For our build philosophy and the story behind this course, see the [main course introduction](#).

Welcome back to Redstone University!

In our last module, we built a physical way to speak to our computer in binary. We have our keyboard, a set of four simple levers. But right now, those levers are connected to nothing. Our machine can't yet *understand* or *do* anything with the numbers we give it. It can hear us, but it doesn't know the language.

In this module, we're going to give our computer a mind. We're going to take a crucial journey into theory to learn the fundamental grammar of all digital logic. This isn't just a Minecraft lesson; this is the language that powers every computer chip ever made.

Welcome to Boolean Algebra.

Lesson 2.1: The Rules of Thought

Key Takeaway: Boolean algebra gives us a precise language for describing and manipulating logical statements, which is the foundation of all digital circuits.

In the mid-1800s, a mathematician named George Boole developed a new kind of algebra. Unlike the algebra you might know from school, where variables like `x` and `y` can be any number, Boole's variables were much simpler. They could only have two possible values: **True** or **False**.

This system, now called **Boolean Algebra**, was initially a mathematical curiosity. But a century later, when engineers started building the first electronic computers with on/off switches, they realized Boole had already invented the perfect mathematical system to describe them.

- **The Core Idea:** We'll treat our Redstone signals as Boolean variables.
- A powered Redstone line is **True**. We'll also call this `1`.
- An unpowered Redstone line is **False**. We'll also call this `0`.

Boolean algebra gives us a set of rules and operators to manipulate these True/False values. These physical operators are called **logic gates**, and they are the bedrock (pun intended) of all computation.

Lesson 2.2: The Core Operators (The Verbs of Logic)

How We Describe Each Gate

For each logic gate, we will start out with visuals and the formal definitions, then move to the truth table and Boolean expression. This will give us a complete understanding of the gate's function.

- **Minecraft Gate:** A screenshot of the gate implemented in Minecraft.
 - **Circuit Diagram:** CircuitVerse diagram of the gate. Note for primitive gates, this will be a single gate with inputs and outputs. For composite gates, we will compose them from the primitive gates.
 - **Formal Definition:** The high-level concept and official terminology.
 - **Symbols:** The common ways this operator is written in logic and programming.
 - **The Rule:** A plain-English sentence describing what the gate does.
 - **Truth Table:** A complete chart defining the gate's behavior. This is the ultimate "source of truth."
 - **Boolean Expression:** The algebraic/logical representation of the gate's output.
 - **Lab & Experiment:** A hands-on test to verify the gate's function against its truth table.
 - **Real-World Connection:** An example of where this logic is used in real technology.
-

A Note on Our Primitives

In the world of computer science, you can build any logic gate from a small set of "primitive" gates. For this course, our primitives are dictated by the physics of Minecraft itself. The game gives us two logical operations right out of the box:

1. **NOT:** A Redstone Torch naturally inverts a signal. This is our primitive NOT gate.
2. **OR:** Redstone Dust naturally merges signals. If any line powering a central wire is ON, the whole wire becomes ON. This is our primitive OR gate.

From these two building blocks, **NOT** and **OR**, we will construct every other logic gate in our computer. This approach shows you how even the most complex digital machines can be built from the simplest possible parts.

While in real-world electronics, gates like NAND or NOR are often used as universal gates due to their efficiency in hardware, we choose NOT and OR for their intuitiveness and direct correspondence to Minecraft's Redstone system.

Operator 1: NOT (The Inverter) - A Minecraft Primitive

Key Takeaway: The NOT gate flips a signal, turning ON to OFF, or 1 to 0. It's the simplest way to create "opposite" logic in a circuit.

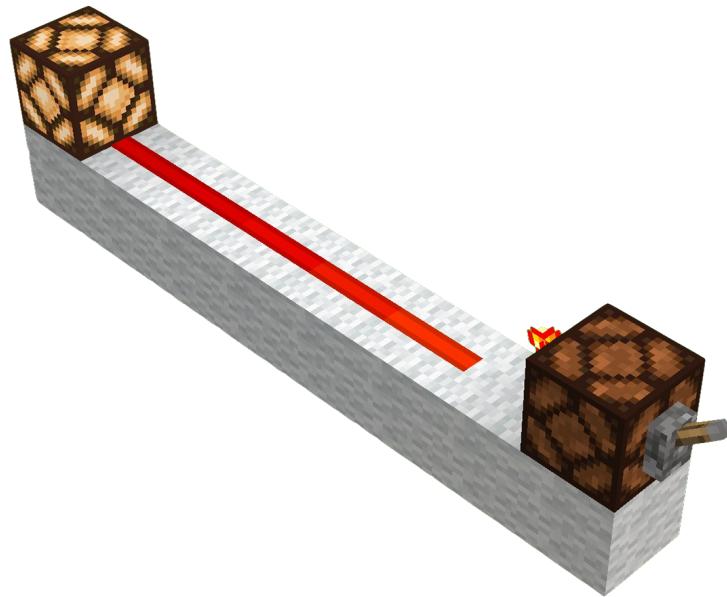


Figure: A single NOT gate with one input and one output, as shown in CircuitVerse.

- **Formal Definition:** The NOT gate, or Inverter, performs **Negation**. It's the simplest possible operation: it takes a single input and outputs its exact opposite.
- **Symbols:** $\neg A$ (logic), $!A$ (programming).
- **The Rule:** If the input is `True`, the output is `False`. If the input is `False`, the output is `True`.
- **Truth Table: NOT Gate**

A	NOT A
0	1
1	0

- **The Boolean Expression:** The output Y is simply $Y = !A$.
- **Lab & Experiment:**



*Figure: The **Redstone Torch** is a purpose-built NOT gate in Minecraft.*

Note: The Redstone Torch itself is a physical NOT gate, but we will add some lamps and dust just to help visualize everything better. Feel free to use a simple torch moving forward if you prefer.

i. Build the circuit as shown in the Minecraft screenshot:

- Place a redstone lamp with a lever on one side to represent input A .
- On the backside of the lamp, place a redstone torch. This is the core component of the NOT gate.
- From the torch, run a line of redstone dust to another redstone lamp representing output Y .

Note: The torch itself is the critical component of the NOT gate. The extra lamps and dust are just for visualization.

ii. Test the circuit:

- Set lever A to ON (1). Observe that the lamp is OFF (0).
- Set lever A to OFF (0). Observe that the lamp is ON (1).

iii. **Verification:** The physical results perfectly match the truth table. You've built a working inverter! The extra lamps and dust we added should help visualize the NOT gate's function, but remember that the torch itself is the core component.

- Real-World Connection:** NOT gates are used everywhere, from creating the oscillating signal in a computer's clock (a "heartbeat") to flipping bits for representing negative numbers, which we'll do in a later module!
- Software Connection:** The NOT operation is used in programming to invert a condition or toggle a flag. For example, in Python:

```

is_on = False
if not is_on:
    print("The device is off.")

```

Here, `not` is the software equivalent of a NOT gate.

Operator 2: OR (The "At Least One" Gate) - A Minecraft Primitive

Key Takeaway: The OR gate outputs 1 if at least one input is 1. It's how we express “either/or” logic in hardware and software.

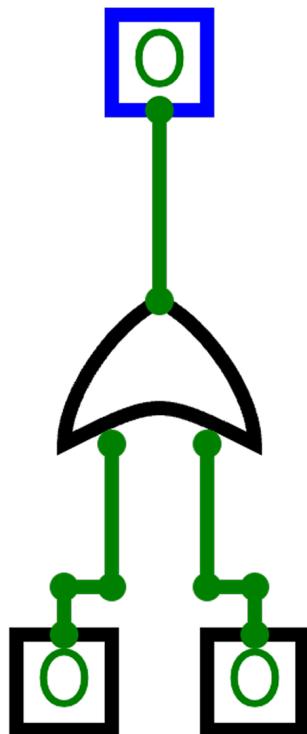


Figure: The OR gate as shown in CircuitVerse.

- **Formal Definition:** The OR gate performs **Disjunction**. Think of it as the optimistic gate—it checks if *at least one* of its inputs is True.
- **Symbols:** $A \vee B$ (logic), `A || B` (programming).
- **The Rule:** The output is `True` if `A` is True, OR `B` is True, or if both are True.
- **Truth Table: OR Gate** | A | B | A OR B | |:---⊕:---⊕:-----⊕| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
- **The Boolean Expression:** The output `Y` is `Y = A OR B`.
- **Lab & Experiment:**

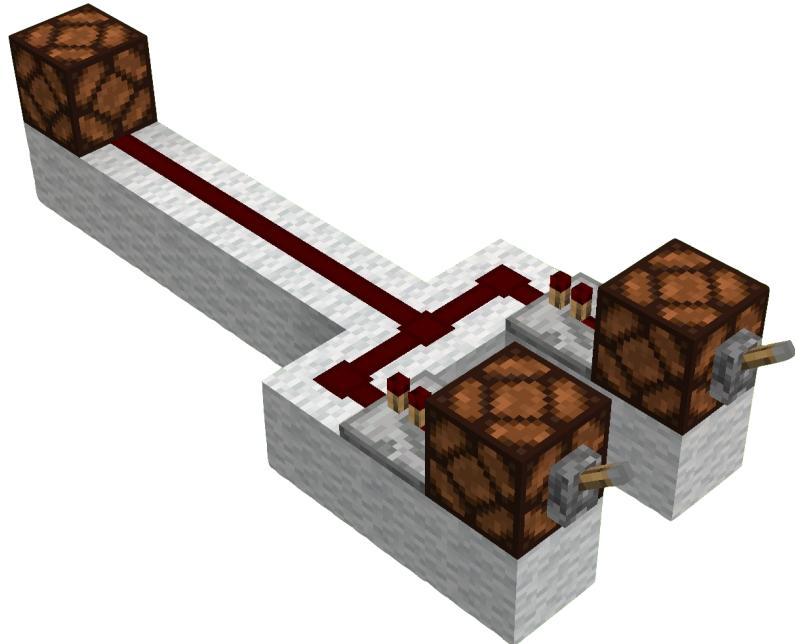


Figure: The classic Minecraft OR gate using Redstone Dust merging.

i. Build the circuit as shown in the Minecraft screenshot:

- Place two redstone lamps with at least one space between them.
- Place a lever on each lamp—these represent inputs `A` and `B`.
- On the other side of each lamp, place a redstone repeater facing away to act as a diode.
- Run dust lines from each repeater and merge them into a single output line.
- Connect this line to another redstone lamp for output `Y`.

ii. **Engineering Note:** When building this OR gate, you might notice that merging dust lines directly from the lamps without repeaters can cause "back-powering"—powering one lamp might light the other, even if its lever is off. The repeaters prevent this by acting as diodes, letting signals flow out but not back in.

iii. Test all four combinations from the truth table (`0,0`, `0,1`, `1,0`, `1,1`).

iv. **Verification:** Confirm the output lamp matches the truth table for each test.

- **Real-World Connection:** A security system might sound an alarm if `FrontDoorSensor=True` OR `BackDoorSensor=True`.

Operator 3: AND (The "Strict" Gate) - Our First Composite Gate

Key Takeaway: The AND gate only outputs 1 if all its inputs are 1. It's how we require multiple conditions to be true at once.

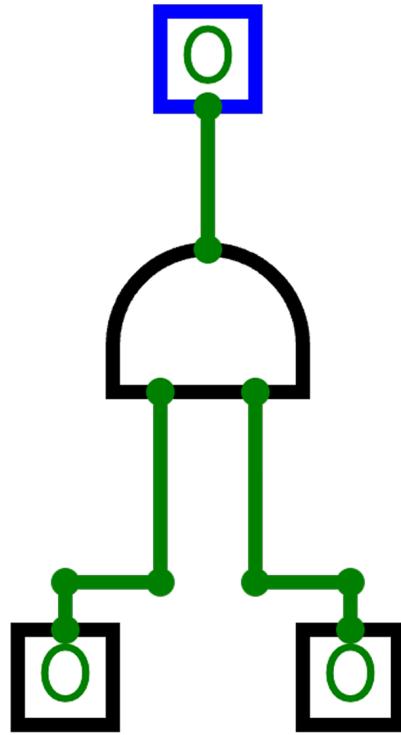


Figure: The standard AND gate symbol in CircuitVerse.

Now we reach our first **composite gate**. Unlike NOT and OR, Minecraft does not give us a single block that performs the AND operation. Instead, we must build it from our primitives. This is a fundamental concept in digital engineering: combining simple components to create more complex functions. Our chosen primitives (NOT and OR) are *functionally complete*, meaning any possible logic function, including AND, can be built from them.

To connect the abstract concept of a gate to our physical build, we will use a consistent visual format. Each composite gate will be introduced with its standard, abstract symbol, which is how engineers represent it in high-level diagrams. This will be followed by a detailed composite diagram showing how to construct it from our primitive NOT and OR gates. In these diagrams, a dashed outline will enclose the group of primitives, visually demonstrating how they work together to become equivalent to the single, abstract gate.

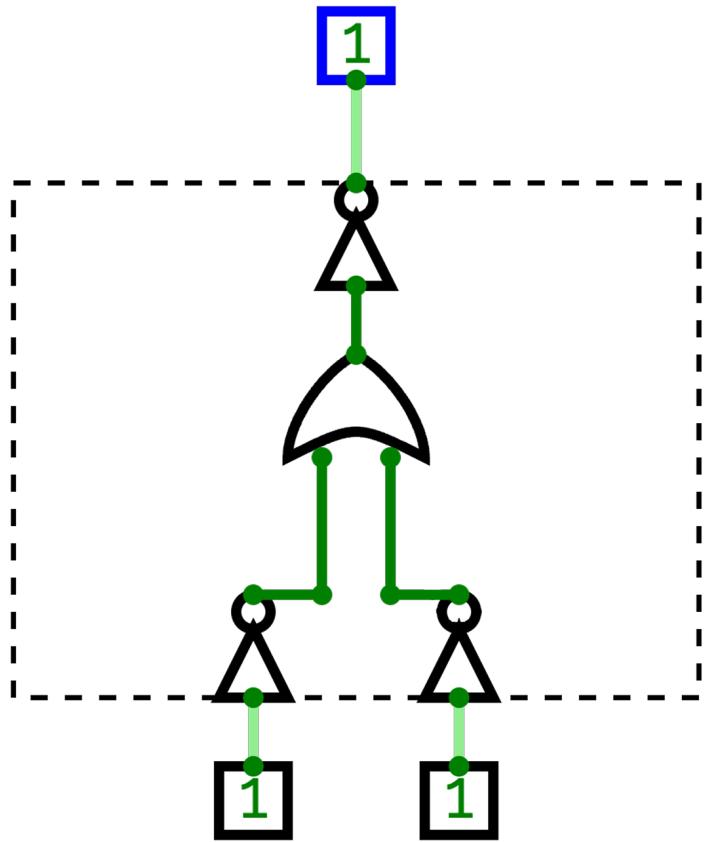


Figure: The AND gate built from NOT and OR gates in CircuitVerse.

- **Formal Definition:** The AND gate performs **Conjunction**. It's the strict gate—output is True only if *all* inputs are True.
- **Symbols:** `A AND B` (logic), `A && B` (programming).
- **The Rule:** The output is `True` only if `A` is True AND `B` is True.
- **Truth Table: AND Gate**

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

- **The Boolean Expression:** The output `Y` is `Y = A AND B`. (Our build uses `!(!A OR !B)`, which we'll prove equivalent in Lesson 2.3.)
- **Lab & Experiment:**

Note on Screenshots and Color Coding: Our Minecraft circuit screenshots use a pseudo-isometric view to show as much of the build as possible. However, it can sometimes be hard to tell if a redstone torch is attached to the backside of a block. To make this clear, any block with a torch on its backside is colored red in the screenshot. Blocks with torches only on top are easy to see, so they use the build's default color unless they also have a backside torch, in which case they're red. For redstone lamps used as inputs (with a lever on one side and a torch or repeater on the other), we can't color code them obviously, but the instructions clearly indicate when a torch is on the backside of one of these input blocks.

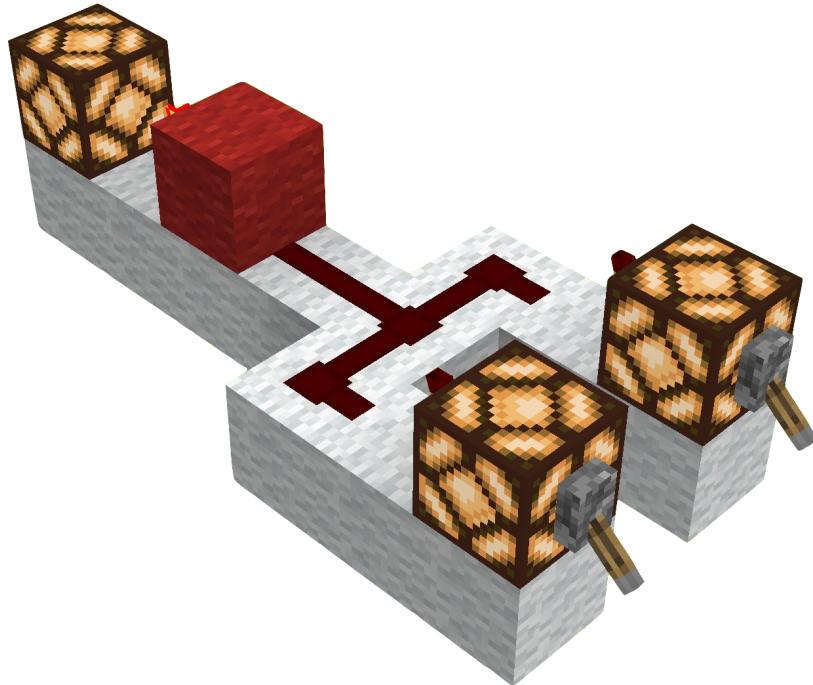


Figure: The verbose AND gate in Minecraft, built as $\neg(\neg A \text{ OR } \neg B)$.

i. Build the verbose version as shown:

- Place two redstone lamps with a lever on the front of each for inputs A and B .
- Attach a redstone torch to the back of each redstone lamp to create the NOT gates for $\neg A$ and $\neg B$.
- Merge these signals to a central point with redstone dust. This creates an OR gate: $\neg A \text{ OR } \neg B$.
- Places a solid block and run the redstone dust into the back of the block.
- Invert this signal by placing a redstone torch on the opposite side of the block. This final NOT gate gives us $\neg(\neg A \text{ OR } \neg B)$.
- Connect the output to a lamp for Y .

ii. Test all four combinations from the truth table ($0, 0$, $0, 1$, $1, 0$, $1, 1$).

iii. **Verification:** The output lamp lights only when both levers are ON.

- **Real-World Connection:** A missile launch might need `TurnKey1=True` AND `PressButton=True`.

Key Takeaway: Boolean laws let us simplify complex circuits and expressions, making our designs more efficient and easier to understand.

Just like $2 + x = x + 2$ in normal algebra, Boolean algebra has laws that let us rearrange and simplify expressions. For us, **a simpler expression means a smaller, faster, and more reliable Redstone circuit**. This is a critical engineering skill.

Boolean Notation: Logical vs Arithmetic

You'll often see logic written using symbols from regular math. For example:

- **AND** is sometimes written as multiplication: $A \cdot B$ or just AB
- **OR** as addition: $A + B$
- **NOT** as an overbar: \bar{A}

For this course, I'll stick with **AND**, **OR**, and **NOT** for clarity, but you'll see this other notation in textbooks and online.

The Laws of Boolean Algebra

Here are the key laws we will be using in our course. There are many more, but these are the most fundamental and useful for circuit design.

- **Identity Law:** $A \text{ OR } 0 = A$ and $A \text{ AND } 1 = A$.
- **Annihilator Law:** $A \text{ OR } 1 = 1$ and $A \text{ AND } 0 = 0$.
- **De Morgan's Law:** This is the superstar. It gives us a way to convert between ANDs and ORs.
 - $!(A \text{ AND } B)$ is the same as $!A \text{ OR } !B$
 - $!(A \text{ OR } B)$ is the same as $!A \text{ AND } !B$
- **Lab: The Proof in Practice** Let's use De Morgan's Law to prove our AND gate design is correct.
 - i. The two redstone torches on the back of our redstone lamps are NOT gates, giving us $!A$ and $!B$.
 - ii. Their signals merge into the central spot, which is an OR gate ($!A \text{ OR } !B$).
 - iii. The final output torch is a NOT gate on that signal. Therefore, the full expression for our circuit is $!(!A \text{ OR } !B)$.
 - iv. Applying De Morgan's Law to the part in the parentheses: $!A \text{ OR } !B$ is the same as $!(A \text{ AND } B)$.
 - v. Substituting that back in, our expression becomes $!(!(A \text{ AND } B))$.
 - vi. The two NOTs ($!!$) cancel each other out, leaving $A \text{ AND } B$. We just proved our physical circuit is correct!

Summary Table: Boolean Laws

Law Name	Example(s)	Description
Identity	$A \text{ OR } 0 = A$ $A \text{ AND } 1 = A$	Leaves value unchanged
Annihilator	$A \text{ OR } 1 = 1$ $A \text{ AND } 0 = 0$	Output is always 1 (OR) or 0 (AND)
Idempotent	$A \text{ OR } A = A$ $A \text{ AND } A = A$	Repeating input doesn't change output
Inverse	$A \text{ OR NOT } A = 1$ $A \text{ AND NOT } A = 0$	Input and its NOT always produce 1 (OR) or 0 (AND)
Commutative	$A \text{ OR } B = B \text{ OR } A$ $A \text{ AND } B = B \text{ AND } A$	Order doesn't matter

Law Name	Example(s)	Description
Associative	$(A \text{ OR } B) \text{ OR } C = A \text{ OR } (B \text{ OR } C)$ $(A \text{ AND } B) \text{ AND } C = A \text{ AND } (B \text{ AND } C)$	Grouping doesn't matter
Distributive	$A \text{ AND } (B \text{ OR } C) = (A \text{ AND } B) \text{ OR } (A \text{ AND } C)$	AND distributes over OR
De Morgan's Laws	$\text{NOT } (A \text{ AND } B) = \text{NOT } A \text{ OR } \text{NOT } B$ $\text{NOT } (A \text{ OR } B) = \text{NOT } A \text{ AND } \text{NOT } B$	Converts between AND/OR with NOT

Functional Completeness: Building with Universal Gates

Universal Gate	To Build a NOT Gate ($\text{!}A$)	To Build an AND Gate ($A \text{ AND } B$)	To Build an OR Gate ($A \text{ OR } B$)
NAND	$A \text{ NAND } A$	$(A \text{ NAND } B) \text{ NAND } (A \text{ NAND } B)$	$(A \text{ NAND } A) \text{ NAND } (B \text{ NAND } B)$
NOR	$A \text{ NOR } A$	$(A \text{ NOR } A) \text{ NOR } (B \text{ NOR } B)$	$(A \text{ NOR } B) \text{ NOR } (A \text{ NOR } B)$

Why does this matter?

For real-world chip designers, this is an incredibly powerful concept. Manufacturing a computer chip is a complex process. Instead of needing separate, specialized machinery to produce AND, OR, and NOT gates, a factory can be optimized to produce just *one* type of gate—like a NAND gate—in massive quantities with extreme reliability and low cost.

Engineers then use the patterns from the table above to wire those identical simple gates together to create all the complex logic they need. The simplicity of manufacturing a single universal gate is a cornerstone of modern, affordable electronics.

Lesson 2.4: The Special Operator – XOR

Key Takeaway: XOR outputs 1 only when its inputs are different. It's essential for circuits like adders and programming tricks.

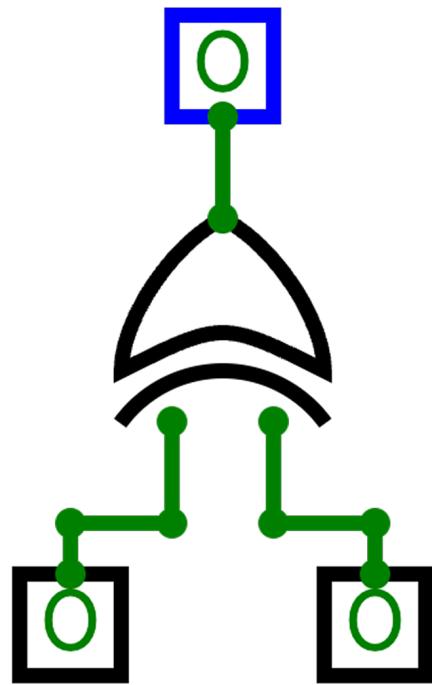


Figure: The standard XOR gate symbol in CircuitVerse.

Like the AND gate, XOR is a composite gate. For all gates we show the abstract symbol used in diagrams as we introduce them, but we will continue our practice of building it from our established primitives. Here is a version of an XOR gate built from OR and NOT, our minecraft primitives.

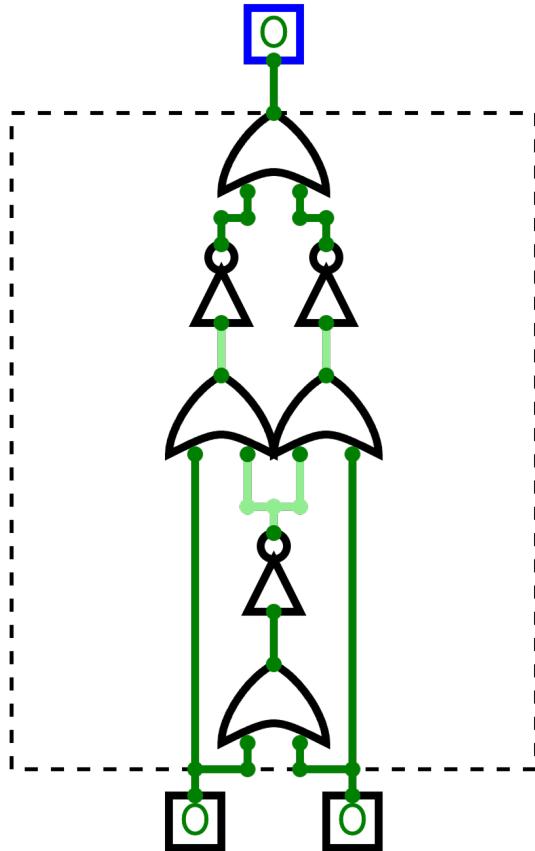


Figure: The XOR gate as shown in CircuitVerse, built from OR and NOT gates.

It's important to understand that this is just one of many ways to build an XOR gate. In Redstone engineering, as in real-world circuit design, there is often no single "correct" answer. Different designs might be bigger but easier to understand, or smaller but more complex. The design above is excellent for visualizing the underlying logic while learning.

- **Formal Definition:** The Exclusive OR (XOR) gate outputs True only when inputs differ.
- **Symbols:** $A \oplus B$ (logic), $A \wedge B$ (programming).
- **The Rule:** The output is `True` if `A` is True and `B` is False, or vice versa; it's `False` if inputs are the same.
- **Truth Table: XOR Gate**

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

- **The Boolean Expression:** $Y = !(A \text{ OR } !(A \text{ OR } B)) \text{ OR } !(B \text{ OR } !(A \text{ OR } B)) .w$

A Note on Design Equivalence: This powerful expression is a direct translation of our circuit diagram. It cleverly uses a shared NOR gate to feed the main logic paths, a common strategy in circuit design for efficiency. We haven't officially introduced NOR gates, but since it is simply a negated OR gate you can look at it as an `OR` gate followed by a `NOT` gate. I went with this design, because it avoids crossing wires while requiring only our primitives.

While it looks very different from the textbook definition, we can prove with a truth table that it is functionally exactly the same. This is a perfect example of how different engineering approaches can lead to the same correct solution. There is always more than one way to build a gate!

- **Lab & Experiment:**



Figure: A composite XOR gate in Minecraft, showing the logic as a combination of our primitive NOT and OR gates.

i. Build the XOR gate as shown in the screenshot:

- Place two redstone lamps with a lever on the front of each for inputs `A` and `B`
- Build the shared `OR` Gate (`A OR B`) by running lines of redstone dust from both inputs `A` and `B` through a repeater each into a single point. This gate will be shared by both inputs.
- Negate the `OR` gate we just built by running the merged line of redstone dust into a solid block. Now place a torch on the opposite side of the block. `!(A OR B)`
- Now we will create our main logic paths with two more negated `OR` gates
 - **Left Path `!(A OR !(A OR B))` :
 - This `OR` gate takes inputs from Input A and from the negated `OR` gate we built in steps 1-3.
 - From the merge point of this `OR` Gate run the redstone in to another solid block.
 - Place a torch on the far side of this second block. The output from this torch is now the entire left half of our boolean expression.
 - **Right Path `!(B OR !(A OR B))` :

- Mirror the left path. Create a third OR gate by running a line of dust directly from input B and another line from the same shared NOR torch's output.
 - Merge these two lines into a third solid block.
 - Place a torch on the far side of this third block. The output from this torch is the complete right half of our boolean expression.
- e. Run a line of dust from each torch of the last two OR gates we built and merge them creating a final OR gate.
- f. Connect this final OR gate merge point to the output lamp Y.
- **Real-World Connection:** XOR is used in adders, error detection, and two-switch light systems (light toggles if one switch changes).

Lesson 2.5: Software Superpowers – The XOR Trick for Programmers

Key Takeaway: XOR is a “secret weapon” in programming. Its reversible, self-canceling property allows for incredibly efficient solutions to common algorithmic problems.

Why is XOR so useful in programming?

The XOR gate has two magical properties that programmers exploit constantly:

1. Any number XORED with itself is zero: $x \wedge x = 0$.
2. Any number XORED with zero is itself: $x \wedge 0 = x$.

Because of these rules, XOR is reversible and "cancels itself out." This allows for brilliant solutions to problems that seem complex at first glance. This is where our hardware knowledge directly translates into writing efficient software.

Let's see it in action with a classic problem from programming interview sites like LeetCode.

Example Problem: The "Single Number"

- **The Challenge:** You are given a list of numbers where every number appears exactly twice, except for one number that appears only once. Find that unique number.
- **Example List:** [4, 1, 2, 1, 2]
- **The XOR Solution:** If you XOR all the numbers in the list together, every number that appears twice will cancel itself out and become zero. The only number left at the end will be the unique one! $4 \wedge (1 \wedge 1) \wedge (2 \wedge 2)$ becomes $4 \wedge 0 \wedge 0$, which is 4.

```
def singleNumber(nums):
    result = 0
    for num in nums:
        result ^= num
    return result
```

Your Turn: The "Missing Number" Challenge

Now that you've seen how the XOR trick works, try applying the same core principle to solve a different, but related, problem.

The Challenge:

You are given a list of numbers that contains every number from 0 to n exactly once, except for one number which is missing. Your task is to find that missing number.

- **Example List:** nums = [3, 0, 1]
- In this example, n would be 3. The full range of numbers should be [0, 1, 2, 3]. The missing number is 2.

The Hint: Think about the two groups of numbers you're dealing with: the list you *have* and the complete list you *should have*. How can you use XOR's self-canceling property to find the single difference between these two groups?

► [Click here for the solution and explanation](#)