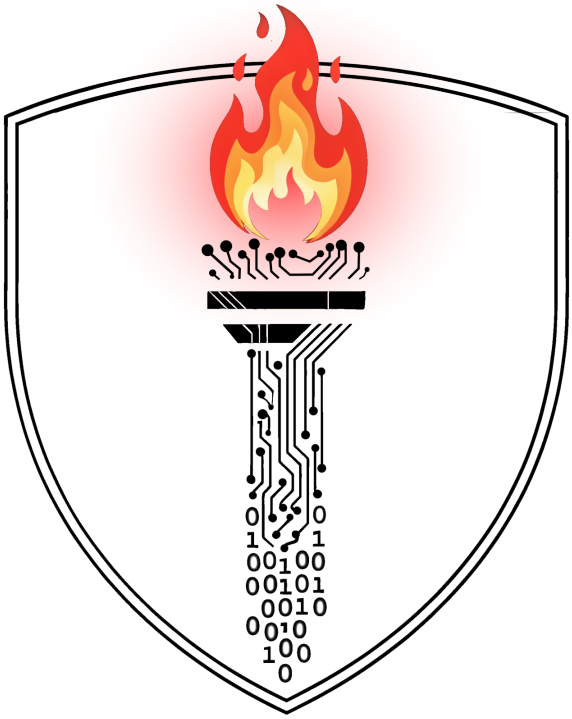

REDSTONE

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 `1`s and `0`s. 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 `1`s and `0`s? 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

CircuitVerse Input Interface

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.

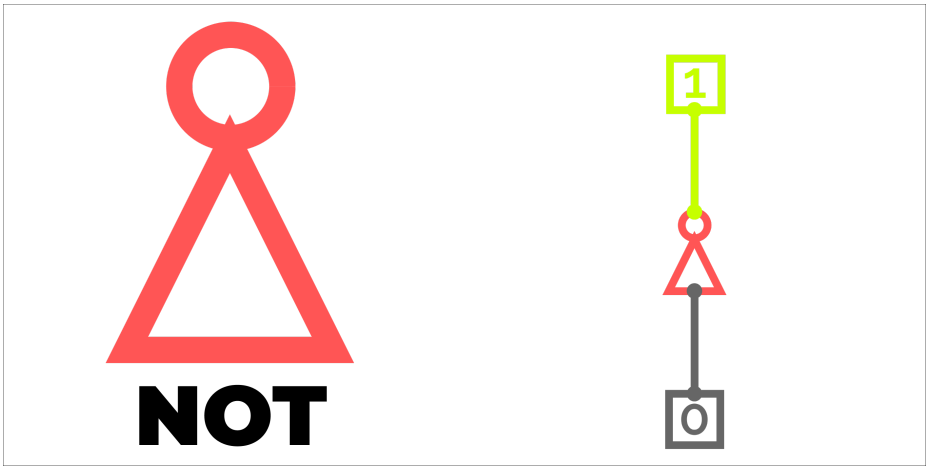

Figure: The abstract symbol for the NOT gate (left) and its function in a basic circuit (right), taking a single input A and producing an inverted output Y.

- **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:** `¬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:**

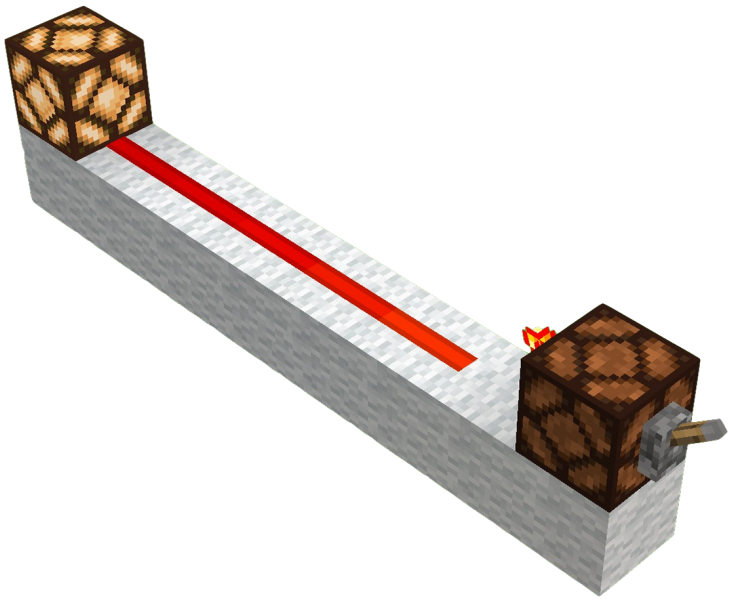

*Figure: The **Redstone Torch** is a purpose-built NOT gate in Minecraft.*

> **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:

   a. Place a redstone lamp with a lever on one side to represent input `A`.
   b. On the backside of the lamp, place a redstone torch. This is the core component of the NOT gate.
   c. 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:

   a. Set lever A to ON (1). Observe that the lamp is OFF (0).
   b. 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 abstract symbol for the OR gate (left) and its function in a basic circuit (right), taking two inputs A and B and producing an output Y that is active if at least one input is active.

- **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 ∨ 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 | 0 | 1 | | 1 | 1 | 1 |
- **The Boolean Expression:** The output `Y` is `Y = A OR B`.
- **Lab & Experiment:**

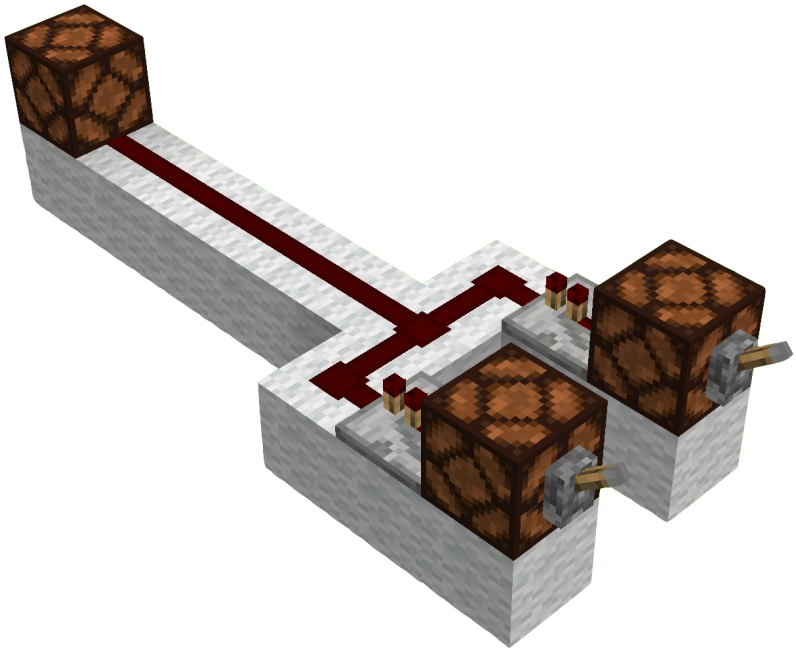

Figure: The classic Minecraft OR gate using Redstone Dust merging.

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

   a. Place two redstone lamps with at least one space between them.
   b. Place a lever on each lamp—these represent inputs `A` and `B`.
   c. On the other side of each lamp, place a redstone repeater facing away to act as a diode.
   d. Run dust lines from each repeater and merge them into a single output line.
   e. 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 abstract symbol for the AND gate (left) and its function in a basic circuit (right), taking two inputs A and B and producing an

output Y that is active only if inputs A and B are both active.

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.

AND Gate Composite in CircuitVerse

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 ∧ 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.

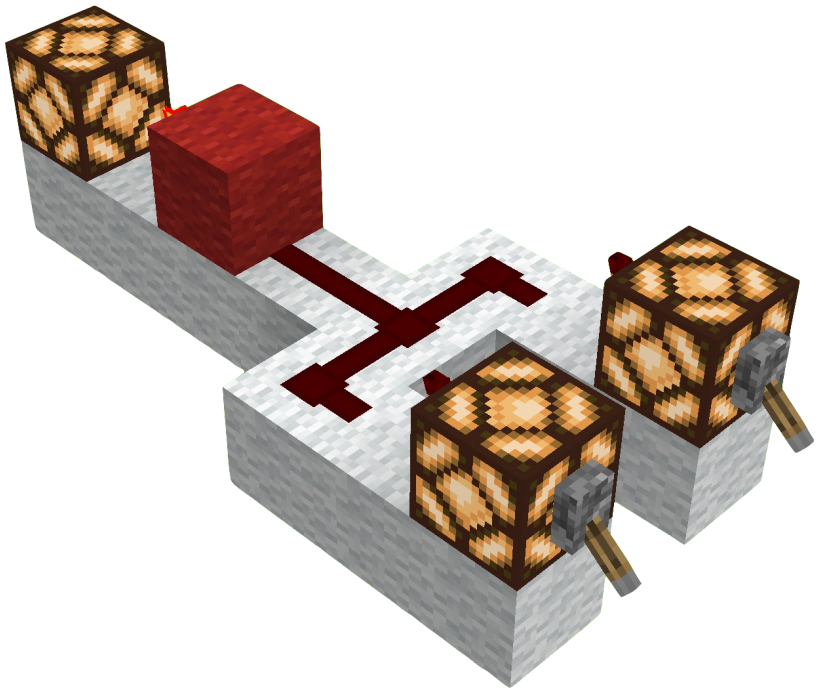

Figure: The verbose AND gate in Minecraft, built as `!(!A OR !B)`.

i. Build the verbose version as shown:

a. Place two redstone lamps with a lever on the front of each for inputs `A` and `B`.
b. Attach a redstone torch to the back of each redstone lamp to create the NOT gates for `!A` and `!B`.
c. Merge these signals to a central point with redstone dust. This creates an OR gate: `!A OR !B`.
d. Places a solid block and run the redstone dust into the back of the block.
e. Invert this signal by placing a redstone torch on the opposite side of the block. This final NOT gate gives us `!(!A OR !B)`.
f. 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`.

Lesson 2.3: The Laws of Logic & The Power of Simplification

> **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 · B` or just `AB`
- **OR** as addition: `A + B`
- **NOT** as an overbar: `Ā`

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 OR 0 = A` and `A AND 1 = A`.
- **Annihilator Law:** `A OR 1 = 1` and `A AND 0 = 0`.
- **De Morgan's Law:** This is the superstar. It gives us a way to convert between ANDs and ORs.
  - `!(A AND B)` is the same as `!A OR !B`
  - `!(A OR B)` is the same as `!A 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 OR !B`).
  iii. The final output torch is a NOT gate on that signal. Therefore, the full expression for our circuit is `!(!A OR !B)`.
  iv. Applying De Morgan's Law to the part in the parentheses: `!A OR !B` is the same as `!(A AND B)`.
  v. Substituting that back in, our expression becomes `!(!(A AND B))`.
  vi. The two NOTs (`!!`) cancel each other out, leaving `A AND B`. We just proved our physical circuit is correct!

Summary Table: Boolean Laws

Law Name	Example(s)	Description
Identity	`A OR 0 = A` `A AND 1 = A`	Leaves value unchanged
Annihilator	`A OR 1 = 1` `A AND 0 = 0`	Output is always 1 (OR) or 0 (AND)
Idempotent	`A OR A = A` `A AND A = A`	Repeating input doesn't change output
Inverse	`A OR NOT A = 1` `A AND NOT A = 0`	Input and its NOT always produce 1 (OR) or 0 (AND)
Commutative	`A OR B = B OR A` `A AND B = B AND A`	Order doesn't matter
Associative	`(A OR B) OR C = A OR (B OR C)` `(A AND B) AND C = A AND (B AND C)`	Grouping doesn't matter
Distributive	`A AND (B OR C) = (A AND B) OR (A AND C)`	AND distributes over OR
De Morgan's Laws	`NOT (A AND B) = NOT A OR NOT B` `NOT (A OR B) = NOT A AND NOT B`	Converts between AND/OR with NOT

Functional Completeness: Building with Universal Gates

Universal Gate	To Build a NOT Gate (`!A`)	To Build an AND Gate (`A AND B`)	To Build an OR Gate (`A OR B`)
NAND	`A NAND A`	`(A NAND B) NAND (A NAND B)`	`(A NAND A) NAND (B NAND B)`
NOR	`A NOR A`	`(A NOR A) NOR (B NOR B)`	`(A NOR B) NOR (A 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 abstract symbol for the Exclusive OR (XOR) gate (left) and its function in a basic circuit (right), taking two inputs A and B and producing an output Y that is active only if the inputs are different.

Like the AND gate, XOR is a composite gate. For all gates we show the asbtract 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.

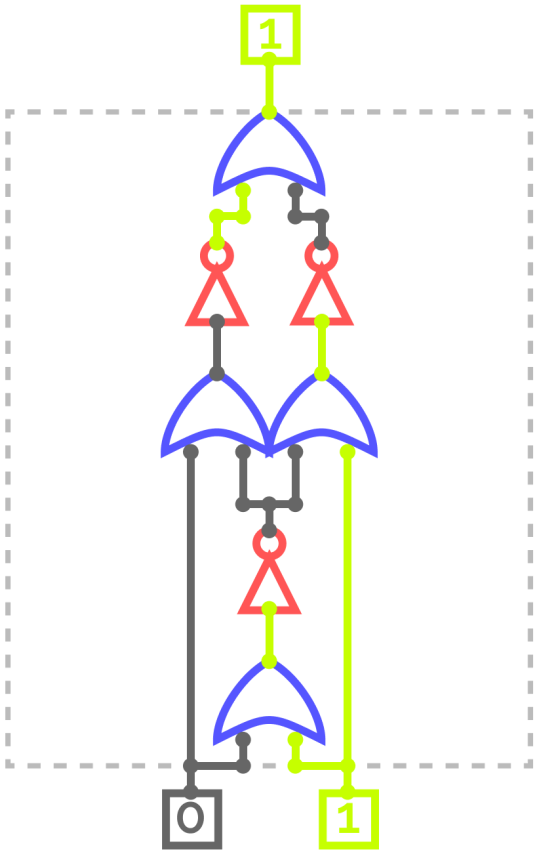

Figure: The XOR gate as shown in CircuitVerse, built from OR and NOT gates.

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 ⊕ B` (logic), `A ^ 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 OR !(A OR B)) OR !(B OR !(A 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:

a. Place two redstone lamps with a lever on the front of each for inputs `A` and `B`

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.

c. 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)`

d. 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 ^ x = 0`.
2. Any number XORed with zero is itself: `x ^ 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 ^ (1 ^ 1) ^ (2 ^ 2)` becomes `4 ^ 0 ^ 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?

> **Solution available in Appendix B: Solutions to Exercises**

Lesson 2.6: The Negated Gates – NAND, NOR, and XNOR

> **Key Takeaway:** Negated gates combine basic operations with NOT. NAND and NOR are “functionally complete”—you can build anything with just one of them!

Operator 4: NAND (The "Not Both" Gate)


NAND Gate in CircuitVerse

Figure: The abstract symbol for the NAND gate (left) and its function in a basic circuit (right), taking two inputs A and B and producing an output Y that is active unless both inputs A and B are active.


NAND Gate (Composite) in CircuitVerse

Figure: A composite NAND gate in CircuitVerse, constructed from AND and NOT gates.

- **Formal Definition:** The NAND gate performs a **NOT-AND** operation—negation of AND.
- **Symbols:** `A NAND B` or `¬(A ∧ B)`.
- **The Rule:** The output is `True` unless both inputs are `True`.
- **Truth Table: NAND Gate**

`A`	`B`	`A NAND B`
0	0	1
0	1	1
1	0	1
1	1	0

- **The Boolean Expression:** `Y = !A OR !B`

> **A Note on De Morgan's Law in Action**: This is one of the most powerful tricks in digital logic. We know that NAND is NOT (A AND B). We also know from De Morgan's Law that NOT (A AND B) is perfectly equivalent to !A OR !B. Our composite AND gate was built as !(!A OR !B). To create a NAND gate, we simply remove the final NOT (the last torch), which leaves us with the physical circuit for !A OR !B. This is a perfect physical proof of a fundamental logic law!

- **Lab & Experiment:**


NAND Gate in Minecraft

Figure: A NAND gate built in Minecraft. The output is off only when both inputs are on.

i. Build the NAND gate:

a. Start by building our composite AND gate from Lesson 2.2.
b. To get the NAND output, remove the final torch.
c. The signal on the dust before that final torch is now your output. Connect this dust to the output lamp for Y. The lamp will now behave exactly like a NAND gate.

ii. Test all four combinations from the truth table.

iii. **Verification:** The output is `0` only when both inputs are `1`.

- **Real-World Connection:** NAND gates are key in hardware (e.g., memory circuits) due to their functional completeness.

Operator 5: NOR (The "Neither" Gate)

NOR Gate in CircuitVerse

Figure: The abstract symbol for the NOR gate (left) and its function in a basic circuit (right), taking two inputs A and B and producing an output Y that is active only if both inputs A and B are inactive.

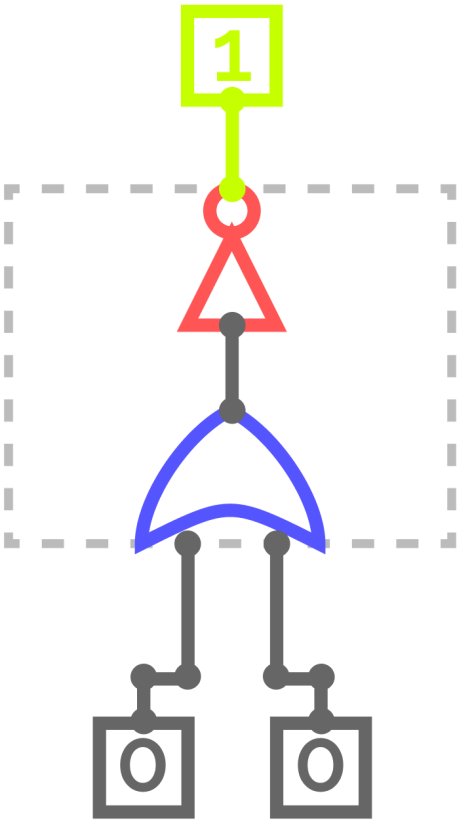

Figure: A composite NOR gate in CircuitVerse, constructed from OR and NOT gates.

- **Formal Definition:** The NOR gate performs a **NOT-OR** operation—negation of OR.
- **Symbols:** `A NOR B` or `¬(A ∨ B)`.

- **The Rule:** The output is `True` only when both inputs are `False`.
- **Truth Table: NOR Gate**

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

- **The Boolean Expression:** The output `Y` is `Y = NOT (A OR B)`.
- **Lab & Experiment:**

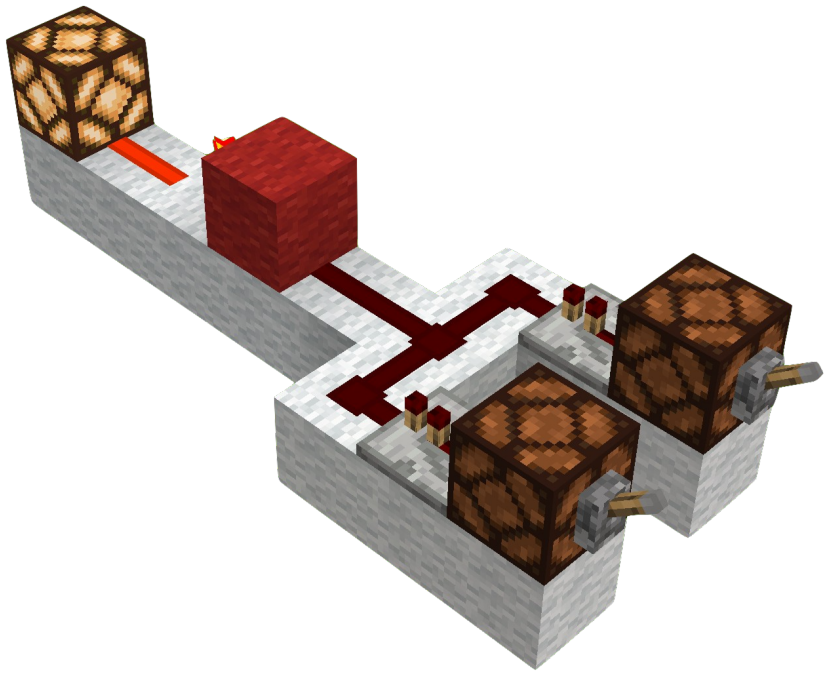

Figure: A NOR gate built in Minecraft. The output is on only when both inputs are off.

i. Build the NOR gate:

   a. Build the OR gate as in Lesson 2.2.
   b. Between the merged dust and the output lamp, place a block with a torch on the output side to invert the signal.
   c. Make sure the dust still connects everything to the output lamp for `Y`.

ii. Test all four combinations from the truth table.

iii. **Verification:** The output is `1` only when both inputs are `0`.

- **Real-World Connection:** NOR gates are used in logic circuits needing a “neither” condition and are also functionally complete.

Operator 6: XNOR (The "Equality Detector")

XNOR Gate in CircuitVerse

Figure: The abstract symbol for the Exclusive NOR (XNOR) gate (left) and its function in a basic circuit (right), taking two inputs A and B and producing an output Y that is active only if the inputs are the same.

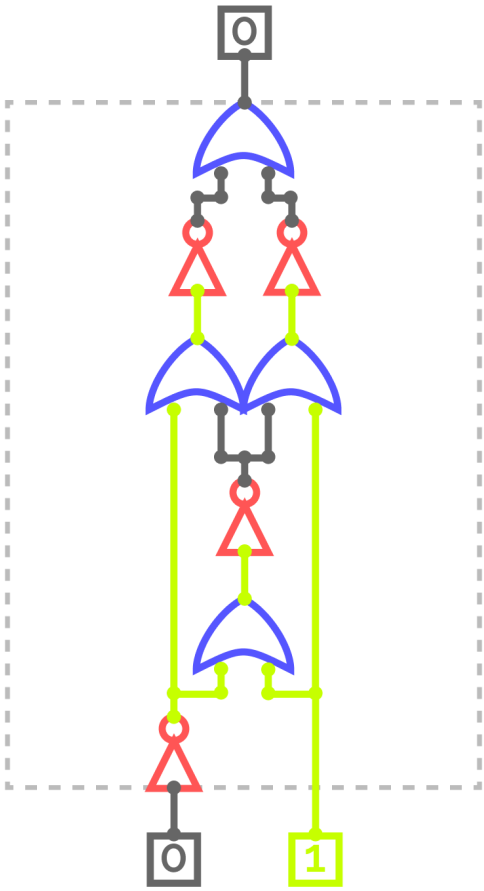

Figure: Composite XNOR gate in CircuitVerse, built by inverting one input to an XOR gate. This demonstrates that XOR(A, NOT B) is equivalent to XNOR(A, B).

- **Formal Definition:** The XNOR gate performs a **NOT-XOR** operation—negation of XOR.
- **Symbols:** `A XNOR B` or `¬(A ⊕ B)`.
- **The Rule:** The output is `True` when inputs are the same (both `True` or both `False`).
- **Truth Table: XNOR Gate** | `A` | `B` | `A XNOR B` | |:---😐:---😐:----------😐 | 0 | 0 | 1 | | 0 | 1 | 0 | | 1 | 0 | 0 | | 1 | 1 | 1 |
- **The Boolean Expression:** `Y = NOT ( !(A OR !(A OR B)) OR !(B OR !(A OR B)) )`
- **Lab & Experiment:**

Figure: XNOR gate in Minecraft, created by adding a NOT gate to one input of our composite XOR gate. This matches the XNOR truth table: the output is on when both inputs are the same.

> **Verifying the Build: A Proof of Equivalence**
>
> We are building our XNOR gate using a fantastic shortcut: an XNOR gate is functionally identical to an XOR gate if you just invert one of its inputs (`XOR(A, !B)`).
>
> How can we prove this using only our primitive gates?
>
> i. The expression for `XNOR(A, B)` is our entire complex XOR expression wrapped in a NOT.
> ii. The expression for `XOR(A, !B)` is our entire complex XOR expression, but with every `B` replaced by `!B` .
> While the direct algebraic proof is incredibly long, we can use a **truth table** to verify it. If you trace all four input combinations for both of these complex expressions, you will find they both produce the exact same final output column: `1, 0, 0, 1` . The trick works perfectly, even with our primitive-only design!

i. **Build the XNOR gate:**

   a. First, build the complete **composite XOR gate** exactly as you did in Lesson 2.4.
   b. Now, modify one of the inputs. We will invert input `B` .
   c. Place a NOT gate on the input line for `B` before it enters the XOR circuit. The easiest way is to move the `B` lever back one block, place a torch on the back of the block the lever is on, and run the signal from that torch into the XOR gate's `B` input.
   d. Ensure the dust from this new NOT gate correctly connects to the rest of the XOR circuit where the `B` input used to be.

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

iii. **Verification:** The output is `1` only when the inputs are the same. You have successfully created an XNOR gate by modifying an XOR gate.

- **Real-World Connection:** XNOR gates are used in equality checks, like comparators in computing.

Lesson 2.7: Module 2 Checkpoint

> **Mini-Summary:** You've learned the core logic gates, their symbols, and how to combine them. You've also seen how Boolean algebra powers both hardware and software problem-solving!

> **NOTE:** Insert summary visual or concept map for all gates.

- **Quiz:**

  i. What is the output of `(1 AND 0) OR (1 XOR 1)`? (Answer: 0)
  ii. If `A=0` and `B=1`, what is `!(A OR B)`? (This is a NOR gate. Answer: 0)
  iii. Which logic gate acts as an "Equality Detector"? (XNOR).

- **Debug Challenge ("Module 2.5"):**

  > In the world download, you'll find a section labeled "Module 2 Debug Challenge." I've built a circuit that is *supposed* to implement the logic `(A AND B) OR C`, but it's giving the wrong output for some inputs! Your mission is to use your knowledge of truth tables to diagnose the mistake in the wiring and fix it.

Module 2 Conclusion

This was a huge module! But you now have the most powerful tool an engineer can possess: a formal language to describe, design, and simplify complex systems. You know the "verbs" of logic, have built them from Minecraft's true primitives, and seen how that physical logic directly empowers elegant software solutions.

What's Next: Now that you can "think in logic," you're ready to build circuits that translate, display, and process information. In the next module, we'll use this newfound language to build our first truly complex and useful machine: a translator that will let our computer speak to us.

> **A Note on the Following Optional Interlude** You have successfully completed Module 2. Congratulations! Before you move on to the next project, we have a special, optional section called an "Interlude." In this module, we focused on building for clarity, making our gates large so the logic was easy to see. The Interlude introduces the art of building for efficiency and compact design. Think of it as your first engineering deep-dive. You can read it now, or you can come back to it at any time. The choice is yours.

Key Terms (Module 2)

- **Boolean Algebra:** A branch of mathematics for working with true/false values (1/0), using operators like AND, OR, and NOT.
- **Logic Gate:** A physical or virtual device that implements a Boolean operation.
- **Primitive Gate:** A basic, indivisible logic gate from which more complex gates are built. In our course, these are NOT and OR.
- **Composite Gate:** A logic gate constructed by combining primitive gates (e.g., an AND gate built from NOT and OR gates).
- **Truth Table:** A chart showing all possible input/output combinations for a logic gate or circuit.
- **Functionally Complete:** A set of gates from which any Boolean function can be built (e.g., just NAND or just NOR).
- **Bitwise Operation:** A software operation that manipulates individual bits of a number.
- **XOR (Exclusive OR):** Outputs 1 if inputs are different; used in both hardware and software for unique logic tricks.

Interlude I: The Art of Compact Design (Optional)

A Note from the Instructor:

Congratulations on finishing Module 2! You've mastered the theoretical foundation of our entire computer.

Before we begin our next major project in Module 3, we have this special, optional section. Think of it as an engineering deep-dive. The goal of Module 2 was to build for **clarity**. This Interlude introduces the art of building for **efficiency**.

You can read it now to prepare for the builds ahead, or you can skip it and come back anytime. In the course we will be using the abstract representation of our logic gates and how you implement them is completely up to you. That's the beauty of black box abstractions, we only care that it provides the interface that is defined in the spec, we don't care HOW it was implemented as long as it works.

Interlude Summary

- **Narrative Beat:** You've mastered the language of logic. Now, let's learn the art of the Redstone engineer: how to shrink those textbook examples into sleek components ready for a real machine.
- **Learning Goals:**
  - Understand the engineering trade-offs between a circuit's size, speed, and readability.
  - Learn common techniques Redstone engineers use to make circuits more compact.
  - Analyze a classic compact AND gate design to see these principles in action.
- **Minecraft Artifact:** A compact version of the AND gate, built and understood.

Introduction

The circuits you built in Module 2 were designed with one goal: **clarity**. They are large and easy to trace so you can see how Boolean logic translates directly into physical blocks.

But when you need to build dozens of gates for a complex component, space becomes a precious resource. This is where engineering comes in. In this appendix, we will explore the philosophy of **compact design**, optimizing our circuits for size and speed.

The Engineering Trade-Off: Size, Speed, and Readability

In Redstone, every design choice is a trade-off. When compacting a circuit, you are usually trading **readability** for **efficiency**.

Factor	Verbose (Educational) Builds	Compact (Practical) Builds
Size / Footprint	Large and sprawling.	Small and dense. Aims to fit the most logic in the smallest area.
Speed / Tick Delay	Often slower due to more components.	Can be significantly faster by minimizing the signal path.
Readability	Very easy to read and debug.	Often difficult to read, making it very challenging to find mistakes.

Your goal is to find the right balance. For learning, verbose is best. For practical builds, compact is essential.

Case Study: The Compact AND Gate

Let's put this into practice by analyzing one of the most classic compact designs in Minecraft. First, recall our verbose AND gate, built to be easy to understand.

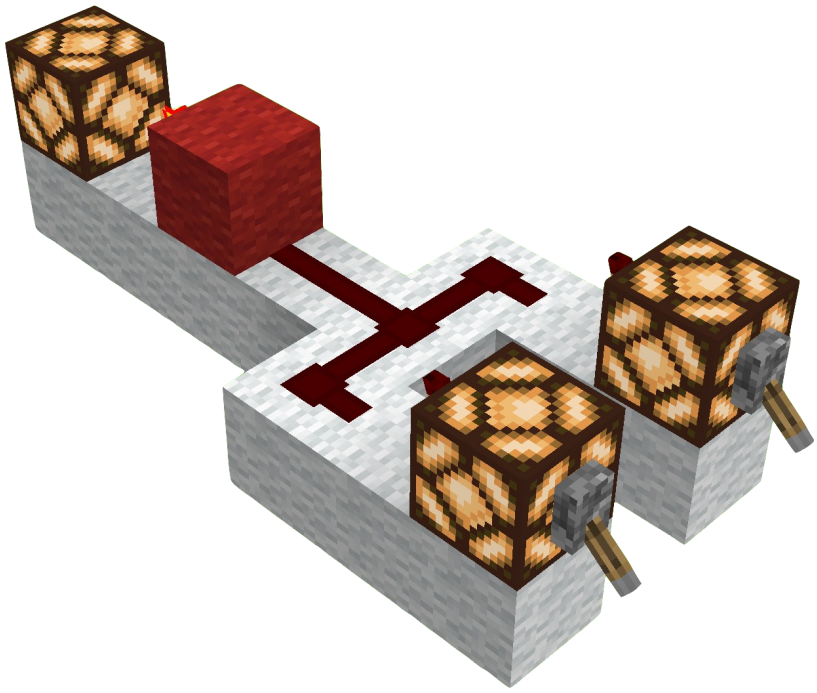

Figure: Our easy-to-read, but large, educational AND gate.

Now, look at the design below. It performs the exact same logic, but in much smaller space. If we remove the lamps we are using to visual input then it is even more compact!

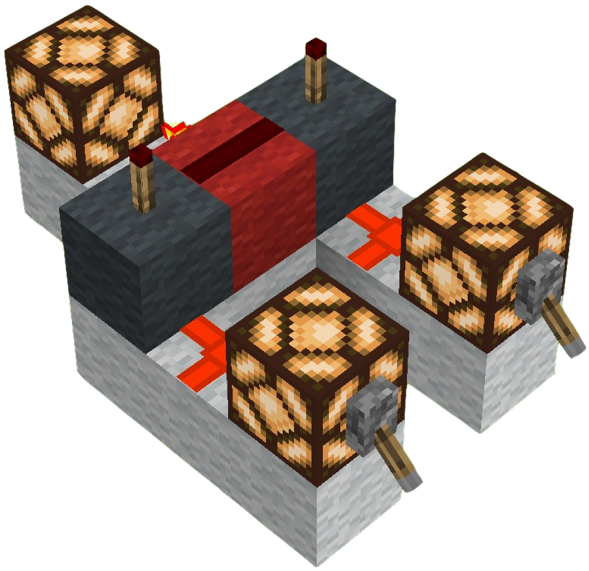

Figure: A classic, space-efficient compact AND gate.

```
Let's see how this works. It's still `!(!A OR !B)`, but the components are cleverly merged.

1.  Place two redstone lamps with a lever on the front of each for inputs `A` and `B`.
2.  Place redstone dust directly behind each lamp.
3.  Directly behind the redstone place a solid block.
4.  Place a torch directly on top of each of the solid blocks to negate both `A` and `B` (`!A` and `!B`).
5.  Between these two blocks place another solid block. For clarity, it can help to make this a different co
6.  Place redstone on top of the middle block which will serve as our OR gate (`!A OR !B`).
7.  On the backside of this middle block, place a redstone torch to negate the signal `(!(!A OR !B))`.
8.  Directly in front of the torch on the backside of the middle block, place your redstone lamp for output
```

Case Study: The Compact NAND Gate

TODO: include the composite and compact figures for NAND

TODO: Explain to the user that they apply the same process that we did in module 2... Negate the final output of the compact AND gate of their choice..

Case Study: The Compact XOR Gate

```
Let's look back at our XOR Gate design from module 2...*TODO*: recall xor build from module 2

<div align="center"><img src="https://raw.githubusercontent.com/fielding/redstone-university/refs/heads/main

*TODO*: Introduce compact version

<div align="center"><img src="https://raw.githubusercontent.com/fielding/redstone-university/refs/heads/main
```

The Build:

```
2.  Build the compact version:
    1.  This more efficient layout uses the same principles but in a smaller space.
    2.  Inputs A and B power blocks that have torches on three sides, creating a complex interaction of NOT
    3.  The final output is taken from the torch at the front. It will only be ON if the inputs are differen
    4.  Connect to an output lamp for `Y`.

3.  Test all four combinations from the truth table (`0,0`, `0,1`, `1,0`, `1,1`).
4.  **Verification:** The output is `1` only when inputs differ.
```

Case Study: The Compact XNOR Gate

TODO: include the composite and compact figures for XNOR

The Build:

TODO: Add build instructions

Conclusion: Your Journey Into Optimization

You now understand the crucial difference between a circuit designed for learning and one designed for practice. This is the first step toward thinking like an engineer.

You do not need to memorize compact designs to complete this course. The verbose builds will work just fine. However, understanding *why* compact designs exist will make you a much better builder.

Explore More in the World Download! TODO: Put together a world showcasing a wide variety of compact logic gate designs and tricks.

This case study is just the beginning. To help you on your journey, the Module 2 world download includes a "Gate Museum" showcasing many different community-tested designs for each logic gate. I encourage you to explore them and use the principles you learned here to understand how they work.

Module 3: Translators & Our First Display

Module Summary

- **Narrative Beat:** We've learned the computer's language. Now let's build a translator so it can talk back to us. A complex translation is often easiest when broken into two simpler steps: first, translating binary to a single idea, then translating that idea into a picture.
- **Learning Goals:**
  - Understand the concepts of decoders and encoders.
  - Apply Boolean logic to a large-scale, modular project.
  - See the direct connection between these components and real-world computer hardware.
- **Lesson Overview:**
  - Lesson 3.1: The Two-Stage Approach
  - Lesson 3.2: The Lab – Building Stage 1 (The 4-to-10 BCD Decoder)
  - Lesson 3.3: The Lab – Building Stage 2 (The ROM and Display Encoder)
  - Lesson 3.4: The Final Connection and The Grand Payoff
  - Lesson 3.5: Module 3 Checkpoint
- **Minecraft Artifact:** A working two-stage translator: a 4-to-10 BCD decoder and a 7-segment display encoder, forming a complete digital display system.

Module Introduction

In the previous modules, you learned how to speak to your computer in binary and how to manipulate those signals with logic gates. But a computer that can only listen isn't very satisfying. We want it to talk back! This is our first large-scale engineering project, and with it comes a new way of thinking.

> **Our New Rule: The Power of Abstraction**
>
> In Module 2, we built every gate from scratch to understand how it worked. From this point forward, we will operate at a higher level of abstraction.
>
> When a diagram or instruction says to "Build an AND gate," **how you choose to build it is now up to you.**
>
> - You can build the verbose, easy-to-read version from Module 2.
> - You can use a smaller, more efficient version from the Interlude.
> - You can design your own!
>
> As long as your component functions according to its truth table, it is a valid build. This freedom is a major step in your journey from student to engineer. The preceding Interlude, **The Art of Compact Design**, gives you the foundation for making these choices.
>
> If you are ever unsure, the verbose builds from Module 2 are guaranteed to work.

Now, let's dive into this module's challenge. You'll build a translator that lets your computer display numbers in a way that humans can instantly recognize. We'll break this down into two manageable stages: first, decoding binary numbers into a single, recognizable idea (a digit), and then encoding that idea into a pattern of lights on a 7-segment display. This modular approach mirrors how real computers handle complex translations and will give you a powerful new tool for your engineering toolkit.

Lesson 3.1: The Two-Stage Approach

When you see the binary `0101` and want to show a "5" on a display, your brain does two things:

1. **Decoding:** Recognize that `0101` *is* the number 5.
2. **Encoding/Mapping:** Recall which segments on a display need to light up to *draw* the shape of a 5.

We'll build a machine that mimics this exact two-stage process:

- **Stage 1: The Binary-to-Decimal Decoder.** This circuit looks at a 4-bit binary input and activates **one, and only one,** of its 10 output lines (one for each digit 0–9).
- **Stage 2: The Display Encoder/Driver.** This circuit takes the active digit line and sends power to the correct segments (a, c, d, f, g, etc.) to draw the right number.

This modular design is a core principle of good engineering. It lets us build, test, and understand each part separately before combining them.

Conceptual Diagram: Two-Stage Translation

```
[4-bit Binary Input]
         |
         v
   [Decoder: 4-to-10]
         |
         v
[One-hot Output (0–9)]
         |
         v
   [Encoder/ROM: 7-segment]
         |
         v
 [7-Segment Display Output]
```

Lesson 3.2: The Lab – Building Stage 1 (The 4-to-10 BCD Decoder)

Goal: Build a circuit that takes a 4-bit BCD input (0–9) and activates one of 10 corresponding output lines. This is a pure application of your Module 2 logic skills.

The Design (On Paper First):

- Let your four input bits be `B3, B2, B1, B0`.
- Let your ten output lines be `L0, L1, ... L9`.
- Each output line is controlled by its own 4-input AND gate.
  - **Logic for L3 (`0011`):** `(!B3) AND (!B2) AND B1 AND B0`
  - **Logic for L8 (`1000`):** `B3 AND (!B2) AND (!B1) AND (!B0)`

The Minecraft Build:

1. **The Bus:** Start with your 4-bit input interface. Create a full 8-line bus by running each of the 4 input lines in parallel, then splitting each one through a NOT gate to create its inverted version. You should have `B3, !B3, B2, !B2, B1, !B1, B0, !B0` all available.
2. **The Logic Array:** Build ten 4-input AND gates. For each gate, tap into the correct four lines from your bus to detect a specific number from 0 to 9.

ASCII Schematic (Conceptual View):

```
      B3  !B3   B2  !B2   B1  !B1   B0  !B0   (8-line Bus)
      |    |    |    |    |    |    |    |
 L0<--+----*----|----*----|----*----|----*---- [4-input AND for `0000`]
 L1<--+----*----|----*----|----*----*----|---- [4-input AND for `0001`]
 L2<--+----*----|----*----*----|----|----*---- [4-input AND for `0010`]
...and so on for L3 through L9.
```

4-to-10 Decoder Mapping Table

Output	Binary Input	Logic Expression
L0	`0000`	`!B3 AND !B2 AND !B1 AND !B0`
L1	`0001`	`!B3 AND !B2 AND !B1 AND B0`

Output	Binary Input	Logic Expression
L2	`0010`	`!B3 AND !B2 AND B1 AND !B0`
L3	`0011`	`!B3 AND !B2 AND B1 AND B0`
L4	`0100`	`!B3 AND B2 AND !B1 AND !B0`
L5	`0101`	`!B3 AND B2 AND !B1 AND B0`
L6	`0110`	`!B3 AND B2 AND B1 AND !B0`
L7	`0111`	`!B3 AND B2 AND B1 AND B0`
L8	`1000`	`B3 AND !B2 AND !B1 AND !B0`
L9	`1001`	`B3 AND !B2 AND !B1 AND B0`

Test your work! Cycle through the inputs 0–9 and make sure the correct single output line (`L0` – `L9`) activates each time.

Troubleshooting Tips:

- If more than one output line is active, double-check your NOT gates and AND gate wiring.
- If no output line is active, make sure all four input bits are connected and that your AND gates are receiving the correct signals.
- Use colored wool or signs to label each line for easier debugging.

> **PLACEHOLDER:** Insert Minecraft screenshot and CircuitVerse diagram of the 4-to-10 decoder build.

Lesson Summary: You've just built a circuit that can recognize any single digit from 0 to 9 in binary. This is an essential skill for translating between human and machine language.

Real-World Connection: The Instruction Decoder

> The circuit you just built is a simplified version of an **Instruction Decoder** in a real CPU. A CPU reads a binary instruction from memory (like `1011` for "ADD"). It feeds this binary code into a decoder just like this one, which activates a single wire that turns on all the circuitry responsible for performing addition.

Lesson 3.3: The Lab – Building Stage 2 (The ROM and Display Encoder)

Goal: Build a circuit that takes one of the 10 active lines from Stage 1 and lights up the correct combination of the 7 display segments.

The Concept: This stage is effectively a **Read-Only Memory (ROM)**. Its "address" is the active line from Stage 1 (`L0` – `L9`), and its "data" is the 7-segment information we "program" into it.

The Design (The "Diode Matrix"):

- Imagine a grid. The 10 input lines from Stage 1 run horizontally. The 7 output lines for our display segments run vertically, crossing over them. We place a connection where a segment needs to be on for a given number.

Visual Aid (Conceptual Grid):

```
          Seg `a`  Seg `b`  Seg `c`  Seg `d`  Seg `e`  Seg `f`  Seg `g`
            |        |        |        |        |        |        |
L0 --------+--------+--------+--------+--------+--------+--------+ (`0`)
L1 --------+--------+--------+--------+--------+--------+--------+ (`1`)
L2 --------+--------+--------+--------+--------+--------+--------+ (`2`)
...and so on for L3 through L9.
```

7-Segment Display Segment Table

Digit	`a`	`b`	`c`	`d`	`e`	`f`	`g`
`0`	X	X	X	X	X	X	
`1`		X	X				
`2`	X	X		X	X		X
`3`	X	X	X	X			X
`4`		X	X			X	X
`5`	X		X	X		X	X
`6`	X		X	X	X	X	X
`7`	X	X	X				
`8`	X	X	X	X	X	X	X
`9`	X	X	X	X		X	X

(X = segment on)

The Minecraft Build:

1. **Layout:** Run your 10 input lines (`L0` – `L9`) horizontally. Run your 7 output lines (`a` – `g`) vertically.
2. **The Connections:** At every intersection where a connection is needed (e.g., L2 needs to power Segment 'a'), place a **Repeater** facing *away* from the horizontal input line and *towards* the vertical output line. The repeater acts as a "diode," ensuring power flows in only one direction.
3. **Programming:** You are physically "programming" the ROM. For each of the 10 input lines, go across and place repeaters on the vertical segment lines that need to be activated for that number.

> **PLACEHOLDER:** Insert Minecraft screenshot and CircuitVerse diagram of the diode matrix/ROM display encoder.

Troubleshooting Tips:

- If a segment doesn’t light up for a certain digit, check that the repeater (diode) is placed at the correct intersection.
- If multiple digits light up segments incorrectly, verify that each input line only powers its intended segments.
- Use temporary Redstone lamps to test each segment output individually.

Lesson Summary: You’ve created a programmable display driver using a physical “ROM.” This is a powerful concept that bridges hardware and software.

Real-World Connection: Read-Only Memory (ROM)

> This "diode matrix" you've built is a simple form of **Read-Only Memory**. The "program," which is the shape of the numbers, is physically burned into the circuit's layout. Old video game cartridges and a computer's BIOS chip worked on this exact principle, with data permanently stored in the hardware's structure.

Software Connection:

> I'll keep this brief, because if you've done any programming, you've used a **lookup table** or hash map. If you find yourself stuck on an interview question, it is worth remembering that the majority of those types problems can be solved naively using a lookup table.

Lesson 3.4: The Final Connection and The Grand Payoff

Now, connect the two stages together. The 10 output lines from your Stage 1 Decoder become the 10 input lines for your Stage 2 Encoder.

Let's Trace the Signal:

1. You set your input levers to `0011` (3).
2. **Stage 1** activates. The AND gate for `(!B3) AND (!B2) AND B1 AND B0` fires. A signal is sent down the single `L3` line. All other 9 lines are off.
3. The `L3` line enters **Stage 2**.
4. The signal on the `L3` line powers the repeaters at the intersections for segments 'a', 'b', 'c', 'd', and 'g'.
5. Those five segment lines light up, and your 7-segment display shows a perfect, glowing **3**.

Lesson Summary: By connecting your decoder and encoder, you've built a complete translation pipeline from binary input to human-readable output. This is a huge leap in making your computer interactive!

> **PLACEHOLDER:** Insert photo or diagram of the final connected system, showing a number displayed.**

Lesson 3.5: Module 3 Checkpoint

- **Quiz:**

  i. What is the main difference between a decoder and an encoder?
  ii. For the number 2 (`0010`), which segments of a 7-segment display should be active?
  iii. In our two-stage design, which stage is responsible for recognizing the binary pattern `1001` ?

- **Challenge:**

> The letter 'H' can be made on a 7-segment display (segments `b` , `c` , `e` , `f` , `g`). If we wanted to add an 11th input line (`LH`) to our Stage 2 Encoder, what would we need to do to make it display 'H'? Describe where you would place the repeaters.
>
> **Hint:** Think about which horizontal and vertical lines need to connect for the 'H' shape. Try sketching the 7-segment display and marking the segments!
>
> **Table for 'H' on 7-Segment Display:**
>
> | Segment | Should be ON for 'H'? |
> |---|---|
> | `a` | |
> | `b` | X |
> | `c | X |
> | `d` | |
> | `e` | X |
> | `f` | X |
> | `g` | X |
>
> Place repeaters at the intersections of the LH line and segments `b` , `c` , `e` , `f` , and `g` .

Module 3 Conclusion

By breaking the problem down into two distinct, logical stages, you've built a highly complex circuit in a way that is easy to understand, build, and debug. You've created a pure **Decoder** and a pure **Encoder/ROM**, two of the most fundamental building blocks in all of digital electronics. This is a massive milestone.

What's Next? In the next module, you'll discover a critical flaw in our simple BCD translator. You'll also learn how real engineers solve it!

Key Terms (Module 3):

- **Decoder:** A circuit that activates one output line based on a unique binary input.
- **Encoder:** A circuit that translates a single input into a coded output (here, segment patterns).
- **ROM (Read-Only Memory):** Hardware that stores fixed data, often used for lookup tables.
- **BCD (Binary-Coded Decimal):** A way of representing decimal digits in binary.
- **7-Segment Display:** An arrangement of LEDs or lamps used to display digits and some letters.

Part II: The Processor Core - Giving Our Machine a Brain

Congratulations on completing Part I! Take a moment to appreciate what you've built. You have a fully functional I/O system: a 4-bit interface to input numbers and a beautiful two-stage display that can show the results. You've mastered the theory of Boolean logic and applied it to a complex, real-world circuit.

But right now, our machine is just a fancy passthrough. It can display a number, but it can't *do* anything with it. It has a mouth and ears, but no brain.

In Part II, we begin to build that brain by focusing on its most critical capability: **arithmetic**.

Our Mission for Part II

This part of the course is a multi-stage story of engineering, debugging, and upgrading. We will not just build a component; we will discover its flaws and systematically improve it until it's powerful and reliable.

- **In Module 4 (The Adder & The "Decoder" Bug),** we'll build our first calculating circuit, the adder. We will immediately discover that our amazing display from Part I has a critical limitation.
- **In Module 5 (The Hexadecimal Upgrade),** we will solve our first bug by teaching our display to speak Hexadecimal, a far more powerful language for our computer.
- **In Module 6 (The "Overflow" Bug & The Carry Bit),** just when we think our system is perfect, we'll push it to its absolute limit and discover a new, more fundamental bug called "overflow," and learn to harness the carry bit to solve it.
- **In Module 7 (The Subtractor),** we'll complete our arithmetic toolkit. Using a brilliant trick called Two's Complement, we will teach our existing adder how to perform subtraction.

By the end of this Part, you will have built a complete, robust, and versatile **Arithmetic Unit**, capable of handling both addition and subtraction for any 4-bit numbers and displaying their results perfectly. This powerful component will become the cornerstone of our final processor.

Let's get started!

Part III: The Processor Core

Excellent work completing Part II. Take a moment to appreciate what you have accomplished. You have engineered a powerful arithmetic unit that can add and subtract, and you've built a robust display system that can handle any result it produces. You have mastered the art of computer mathematics.

But a processor is more than just a math machine; it's also a *logic* machine. We've built AND, OR, and XOR gates, but they're not yet part of our main processor. The theme for Part III is to finally assemble all of our computational components into the single, unified, controllable brain of our computer: the **Arithmetic Logic Unit (ALU)**.

Our Mission for Part III

This part is focused on the grand assembly of our processor's core. We will build the final control systems and then forge everything into our most complex component yet.

- **In Module 8 (The Multiplexer),** before we can build the ALU, we must first build its "steering wheel." We'll learn about and construct a Multiplexer, a crucial digital switch that allows us to choose between multiple different inputs.
- **In Module 9 (The ALU),** this is the capstone project for our processor. We will bring all our previous work, the adder/subtractor and the logic gates, into one place and use our new Multiplexer to build a complete, multi-function ALU that can be commanded to perform a wide variety of operations.

By the end of this Part, the brain of our computer will be complete. We will have built the single most important component in any CPU, setting the stage for the final act: bringing it to life.

Let's get started with Module 8 and build our digital switch.

Part IV: Creating an Automated Computer

Incredible work on completing Part III. Our machine is now truly impressive. It has a powerful, versatile Arithmetic Logic Unit that can perform multiple types of calculations on command. We have built a genuine, manually-operated processor.

But a computer is more than a processor. It doesn't wait for a human to flip levers for every single step. A true computer can follow a list of instructions, a program, all on its own.

In Part IV, we give our machine a soul. The theme for this part is **Automation**. We are going to build the final architectural components that separate a static calculator from a dynamic, living computer. We will give it a memory to hold its thoughts and a heartbeat to drive it forward.

Our Mission for Part IV

This part will see us construct the final pieces of the puzzle and assemble them into a single, cohesive, self-running system.

- **In Module 10 (Memory),** we will tackle the concept of "state." We will build circuits called latches that can remember a value, giving our processor a "scratchpad" to store its results. This is the foundation of computer RAM.
- **In Module 11 (The Grand Assembly - Automation),** we will build a clock to provide a steady pulse and a Program Counter to automatically step through a sequence of hard-coded instructions. We will take our hands off the levers and watch our creation execute a program for the first time.

By the end of this Part, you will have achieved the ultimate goal of this course: you will have orchestrated a collection of simple components into a machine that can run a program without your intervention.

Let's begin Module 10 and give our computer a memory.

Part V: Post-Graduate Studies - Advanced Engineering

Congratulations, graduate of Redstone University! You have successfully completed the core curriculum. You have designed and built a fully operational, programmable 4-bit computer from scratch. You understand its number system, its logic, its processor, its memory, and the clock that brings it all to life. This is a monumental achievement.

The main course is over, but for those who are hungry for a greater challenge, the university offers a post-graduate program.

In Part V, we will tackle an advanced engineering problem that we sidestepped earlier for the sake of efficiency. This bonus content is designed to stretch your skills and show you the kind of complexity required to make computers perfectly align with human expectations.

Our Mission for Part V

This special section contains a single, challenging module that will test everything you've learned.

- **In Module 12 (The "Real World" Display),** we will finally solve the problem we encountered back in Module 4: how to display a number like "13" using two separate decimal digits. We chose the elegant programmer's solution of Hexadecimal, but now we will build the complex engineer's solution used in real-world calculators and digital clocks: the Double Dabble algorithm.

This final module is not for the faint of heart. It is a true capstone project that will result in the most "human-friendly" version of our computer. It's the perfect challenge for those who looked at their completed computer and asked, "What's next?"

Welcome to advanced studies. Let's dive into Module 12.

Appendix B: Solutions to Exercises

Solution for #### Lesson 2.5: Software Superpowers – The XOR Trick for Programmers

---

The Logic:

The core idea is to XOR all the numbers that *should* be in the list against all the numbers that *are* actually in the list.

1. First, we calculate the XOR sum of the complete sequence of numbers from 0 to `n`. For our example `[3, 0, 1]`, `n` is 3, so this would be `0 ^ 1 ^ 2 ^ 3`.
2. Next, we calculate the XOR sum of the numbers in the list we were given: `3 ^ 0 ^ 1`.
3. If we XOR these two results together, all the numbers that are present in both lists will pair up and cancel out, leaving only the number that was missing from the input list.

`(0 ^ 1 ^ 2 ^ 3) ^ (3 ^ 0 ^ 1)` can be rearranged as `(0^0) ^ (1^1) ^ (3^3) ^ 2`, which simplifies to `2`.

The Python Code:

```
def missingNumber(nums):
    n = len(nums)
    expected_xor_sum = 0
    for i in range(n + 1):
        expected_xor_sum ^= i

    actual_xor_sum = 0
    for num in nums:
        actual_xor_sum ^= num

    return expected_xor_sum ^ actual_xor_sum
```