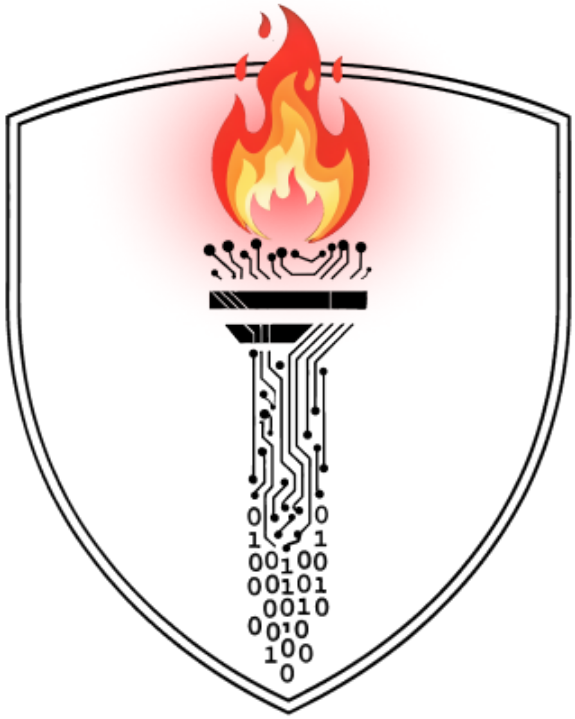

Welcome to Redstone University!

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 the first official part of our Redstone University curriculum! The grand goal is to build a complete computer, but like any great project, we must begin by laying a solid foundation. The theme for this part is the **Human-Computer Interface**: we are going to build the essential components that allow us, as humans, to communicate with our digital machine.

By the end of Part I, our computer won't be able to think for itself yet, but we will have a fully functional input and output system. We'll have a way to give it numbers and a way for it to show us numbers back in a format we can instantly recognize.

Our Mission for Part I

We will tackle this in three distinct modules, moving from the physical to the theoretical, and back to a large-scale physical application.

- **In Module 1 (The Input Register),** we will build our "keyboard." It's a simple set of levers that will allow us to input numbers into our machine using the computer's native language: binary.
- **In Module 2 (Boolean Algebra),** we will take a crucial detour into theory. This is the most important "lecture" in the entire course. We will learn the fundamental grammar of all digital logic—the rules of NOT, AND, OR, and XOR that govern every circuit we will ever build.
- **In Module 3 (Decoders & Displays),** we will take our new theoretical knowledge and immediately apply it to our biggest challenge yet: building a way to display our input and output. We'll engineer a sophisticated two-stage translator that converts the computer's binary into a human-readable number on a beautiful 7-segment display.

This first part of the course is designed to give you an incredibly satisfying payoff. You will start with nothing but switches and end with a device that feels intelligent. The concepts you learn here are the bedrock (pun intended) upon which everything else will be built.

> **Note about the chosen course progression:** I personally found myself wanting to build a 7-segment display to verify that everything past this point in the course was working correctly. I wanted to see the numbers I was entering, and I wanted to see the results of my calculations. This is why we start with the input register and then immediately build the display. It gives us a tangible goal to work towards, and it makes the abstract concepts of binary and logic feel real and rewarding.
>
> This also follows a personal belief about learning to program: The quicker you can get something moving on screen, the more motivated you will be to keep going. I want you to have that immediate sense of progress and accomplishment.

Let's get started with Module 1.

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

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 keyboard 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 Register
  - Lesson 1.3: Drills & Games – Strengthening Your Binary Intuition
  - Lesson 1.4: Module 1 Checkpoint
- **Minecraft Artifact:** A working 4-bit input register (keyboard) 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 construction project, we must begin by laying a solid foundation. The very first thing we need is a way to communicate with our machine. We need a way to give it information.

In this first module, we're going to build our computer's "keyboard." It won't have letters like a normal keyboard. Instead, it will have a set of simple switches that let us speak the computer's one and only native language: `binary`.

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**
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 Register

It's time to stop talking and start building. A **register** is a fundamental piece of computer hardware that holds a single piece of data. Our 4-bit register will be our simple keyboard, allowing us to manually input any number from `0` to `15`.

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 register, I recommend a redstone lamp. It doubles as a visual indicator of the current state of each bit.

The Build Guide

1. Find a nice open, flat area on our campus. This is your personal workbench.
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.

Minecraft Register Example

Figure: The register 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.

CircuitVerse Register Example

View on CircuitVerse

CircuitVerse Register

Figure: The same 4-bit register, 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 a register that holds a 4-bit binary number.

Don't worry, we will be building more interesting circuits very soon. We will be using this register to input numbers into our computer throughout the entire course, so while it might not be impressive looking, it is a crucial part of our computer's architecture.

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 with your register will make binary numbers feel as natural as decimal. The more you play, the faster you'll get!

Drill 1: Binary to Decimal

- **Goal:** What decimal number is `1011`?
- **Action:** Go to your register 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

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 register. 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 register could hold? (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

The `4`-bit register you just built is a real, albeit simplified, computer component. When you see a computer advertised as having a "64-bit processor", it means its internal registers (the spaces inside the chip where it does its most immediate work) are `64` bits wide. They are just like your device, but with `64` "levers" instead of just `4`. This allows them to work with incredibly large numbers in a single step! A `64`-bit register can hold a number larger than `18` quintillion.

In real CPUs, these registers are used to store numbers, addresses, and even instructions. When you write code in assembly language, you are often moving values directly in and out of these registers. Each bit in a register is like a physical wire or lever that can be `ON` or `OFF`, just like your Minecraft build.

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 "wire" of the number one by one.

```
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 in your register!

Extra: Registers are the foundation for fast calculations in CPUs, and bitwise tricks are used everywhere in high-performance code, cryptography, and even graphics.

Module 1 Conclusion

Fantastic work! You've now mastered the most fundamental concept in all of computing: how information is physically stored 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 and make our machine think.

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.

Operator 1: NOT (The Inverter)

> **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.

- **Minecraft Gate:**

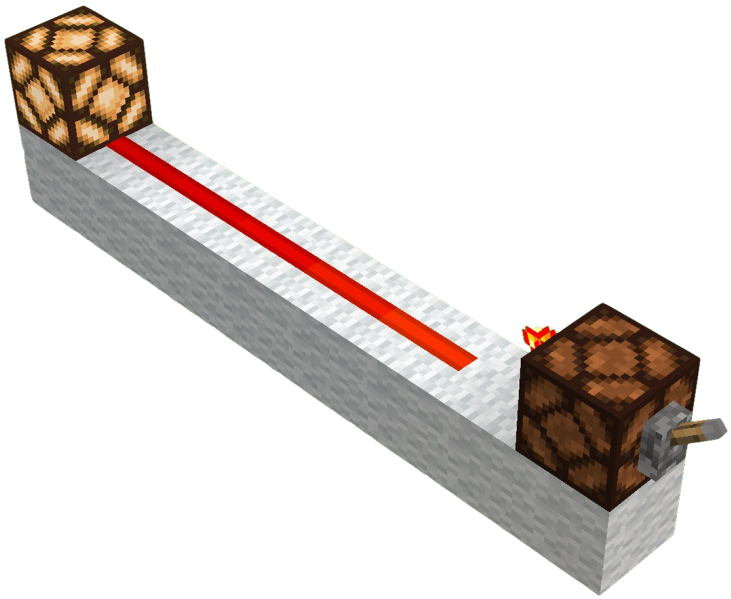

*Figure: The **Redstone Torch** is a purpose-built NOT gate in Minecraft.*

- **Circuit Diagram:**

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:** `¬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:**

> **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)

> **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.

- **Minecraft Gate:**

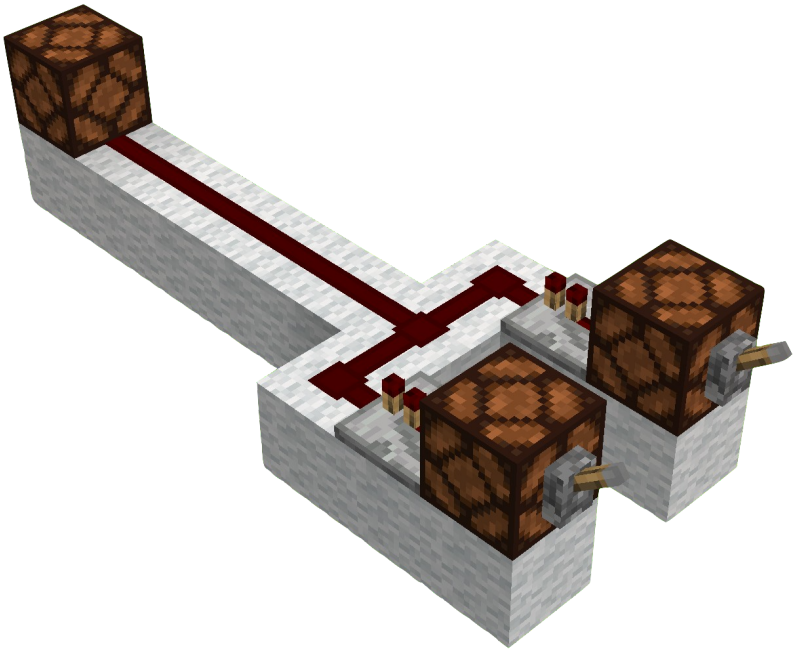

Figure: The classic Minecraft OR gate using Redstone Dust merging.

- **Circuit Diagram:**

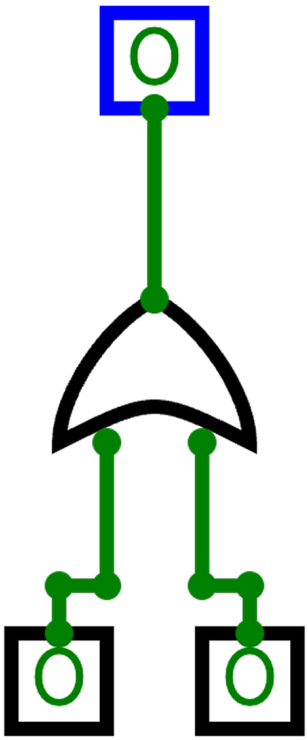

Figure: The OR gate as shown in CircuitVerse.

- **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 ∨ 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:**

    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)

> **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.

- **Minecraft Gate: Our First Composite Gate**

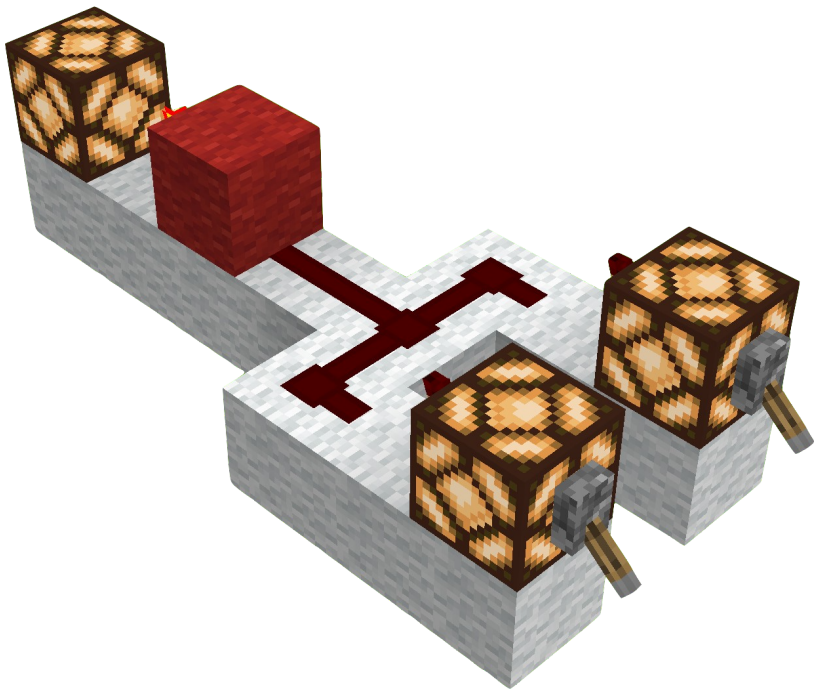

Figure: The verbose AND gate in Minecraft, built as `!(!A OR !B)`.

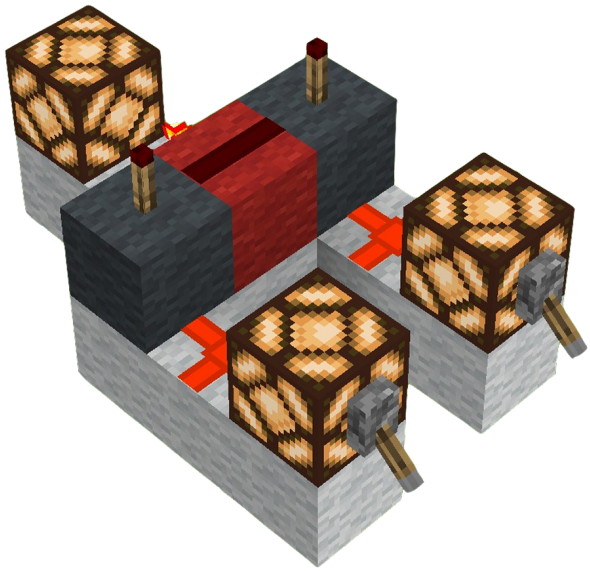

Figure: The compact AND gate in Minecraft, using a more efficient layout.

- **Circuit Diagrams:**

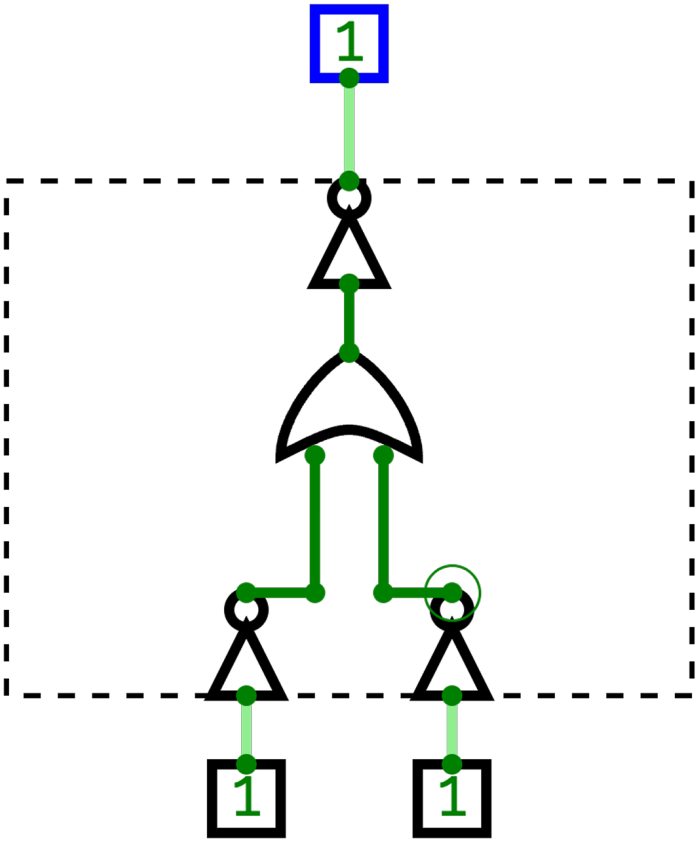

Figure: The AND gate built from NOT and OR gates in CircuitVerse.

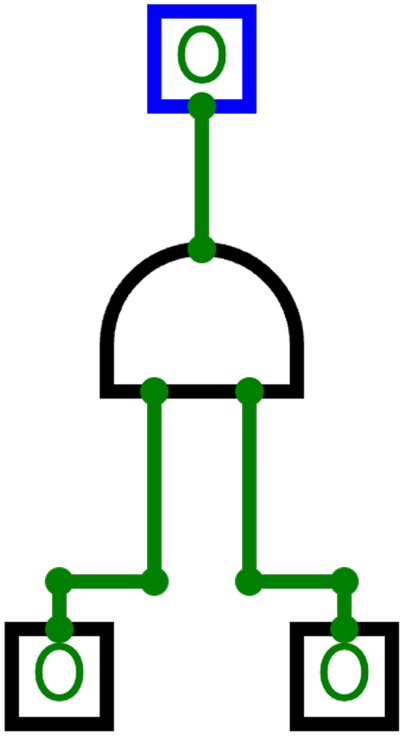

Figure: The standard AND gate symbol in CircuitVerse.

- **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:**
    i. Build the verbose version as shown:
        a. Place two levers for inputs `A` and `B`.
        b. Attach a redstone torch to each input block to create `!A` and `!B`.
        c. Merge these signals into a central block with redstone dust (an OR gate: `!A OR !B`).
        d. Place a torch on the central block to invert the signal, giving `!(!A OR !B)`.
        e. Connect the output to a lamp for `Y`.
    ii. Build the compact version:
        a. Place two levers for `A` and `B`.
        b. Use a similar layout but minimize space (see screenshot).
        c. Connect to an output lamp.
    iii. Test all four combinations from the truth table (`0,0`, `0,1`, `1,0`, `1,1`).
    iv. **Verification:** The output lamp lights only when both levers are ON.
- **Real-World Connection:** A missile launch might need `TurnKey1=True` AND `PressButton=True`.

Why are we building the AND gate this way? In this course, we're treating AND as a key concept, but in Minecraft, we're making it from NOT and OR gates. This shows you how all logic gates can come from a few basic parts. Try building both the detailed (verbose) and compact versions to see how they work!

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.

- **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 side torches are NOT gates on our inputs, giving us `!A` and `!B`.
  ii. Their signals merge into the central block, which is an OR gate (`!A OR !B`).
  iii. The final output torch is a NOT gate on that signal.
  iv. Therefore, the full expression for our circuit is `!(!A OR !B)`.
  v. Applying De Morgan's Law to the part in the parentheses: `!A OR !B` is the same as `!(A AND B)`.
  vi. Substituting that back in, our expression becomes `!(!(A AND B))`.
  vii. The two NOTs (`!!`) cancel each other out, leaving `A AND B`. We used formal logic to prove 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

Law Name	Example(s)	Description
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 Table

Gate Used Alone	Can Build...	Example Construction

NAND	NOT, AND, OR, XOR	`NOT A = A NAND A`
		`AND = (A NAND B) NAND (A NAND B)`
		`OR = (NOT A) NAND (NOT B)`

NOR	NOT, AND, OR, XOR	`NOT A = A NOR A`
		`AND = (NOT A) NOR (NOT B)`
		`OR = (A NOR B) NOR (A NOR B)`

Any logic circuit can be built using just NAND or just NOR gates!

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.

- **Minecraft Gate:**

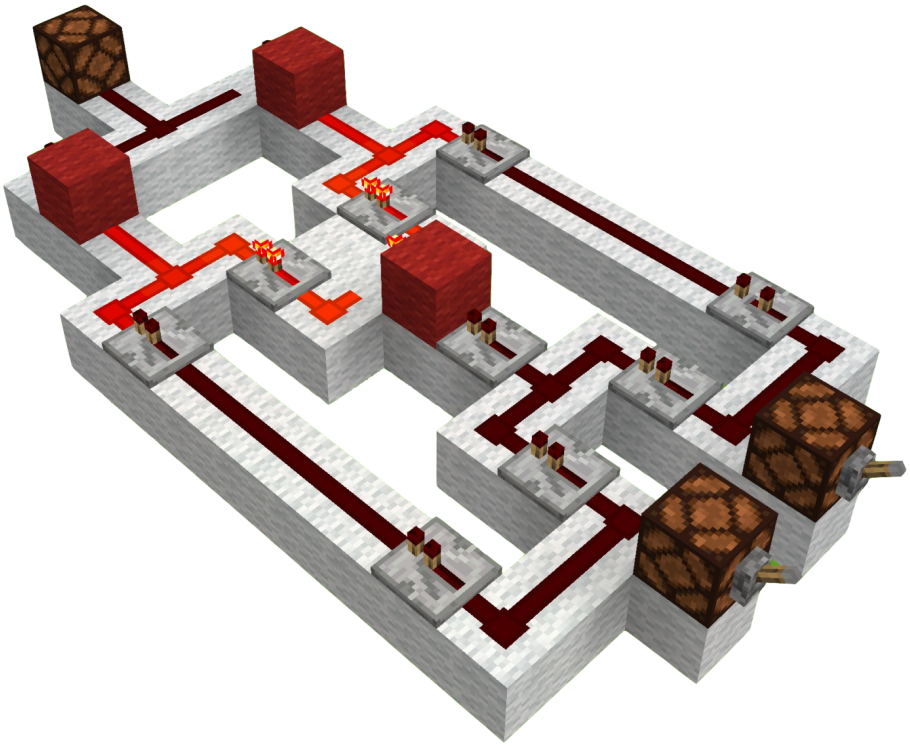

Figure: A composite XOR gate in Minecraft, showing the logic as a combination of AND, OR, and NOT.

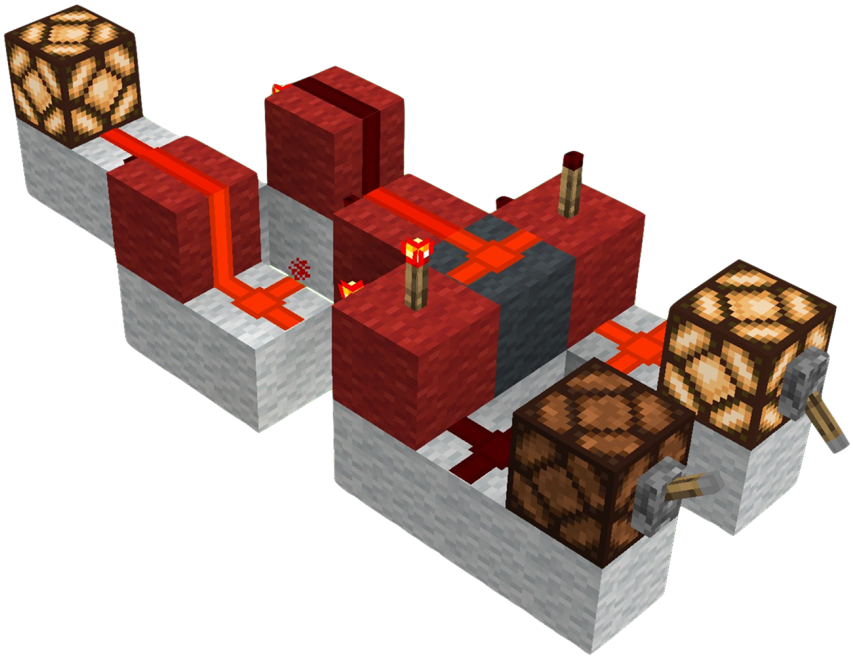

Figure: A compact XOR gate built in Minecraft. The output is on only when the two inputs are different.

- **Circuit Diagram:**

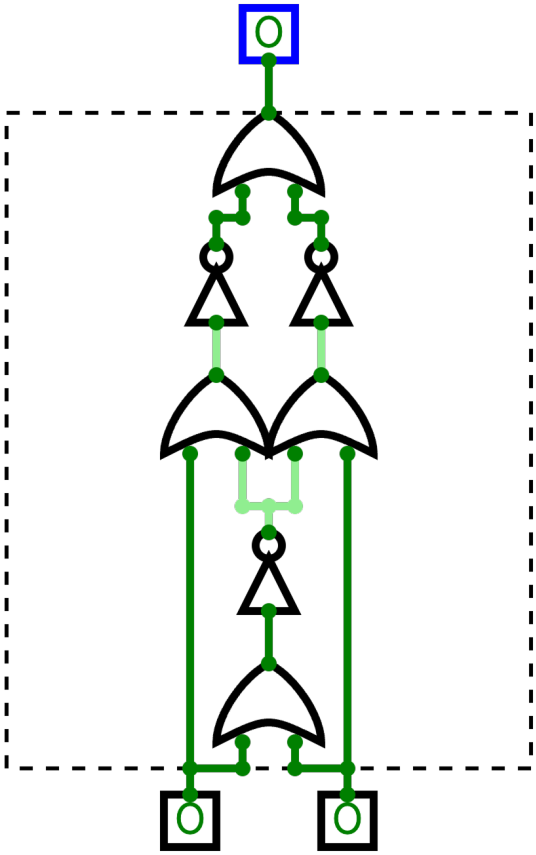

Figure: The XOR gate as shown in CircuitVerse, built from AND, OR, and NOT gates.

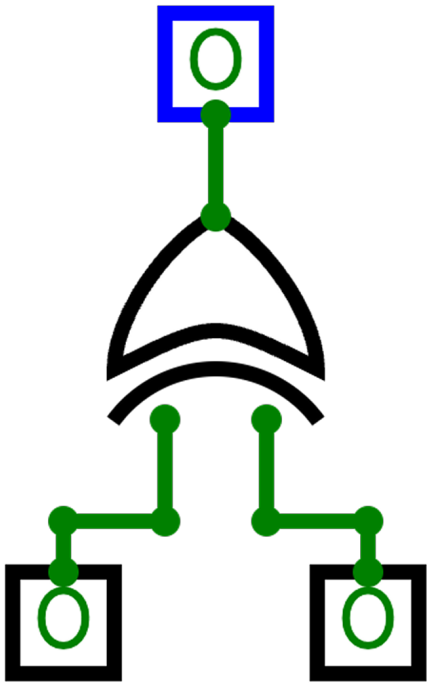

Figure: The standard XOR gate symbol in CircuitVerse.

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

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

a. Set up two levers for inputs `A` and `B`.
b. Construct the gate as shown in the screenshot. (*updated this*)
c. Connect to an output lamp for `Y`.

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 inputs differ.

- **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? Because it's reversible and "cancels itself out." For example, `a ^ a = 0` and `a ^ 0 = a`. This property lets you do things like swap two variables without a temporary variable, or find the one unique number in a list where every other number appears twice.

> **LeetCode Connection:** XOR is a favorite in programming interviews. Here's a classic LeetCode problem and its solution using XOR:
>
> **The "Single Number" Problem:**
>
> - **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 together, the pairs cancel themselves into nothing, leaving only the unique number! `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
> # singleNumber([4, 1, 2, 1, 2]) returns 4
> ```

This is where our hardware knowledge directly translates into writing brilliant, efficient software. 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`.

These two rules allow for an incredibly elegant solution to a whole class of programming interview problems on sites like LeetCode.

- **Real-World Connection:** XOR gates are used in digital circuits for error detection (parity checks), cryptography, and even in RAID storage systems to recover lost data.

- **Software Connection:** XOR is used in programming for toggling bits, finding unique elements, and implementing simple encryption.

The "Missing Number" Problem:

- **The Challenge:** You have a list containing every number from 0 to `n`, except one is missing. Find the missing number.
- **The XOR Solution:** You can XOR all the numbers you *expect* to see (0 to n) with all the numbers you *actually* see in the list. The number that doesn't have a pair is the one that's missing.

This is a powerful bridge between hardware and software. The simple "difference detector" we built in Minecraft is the logical foundation for solving complex algorithmic problems with extreme efficiency.

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)

- **Minecraft Gate:**

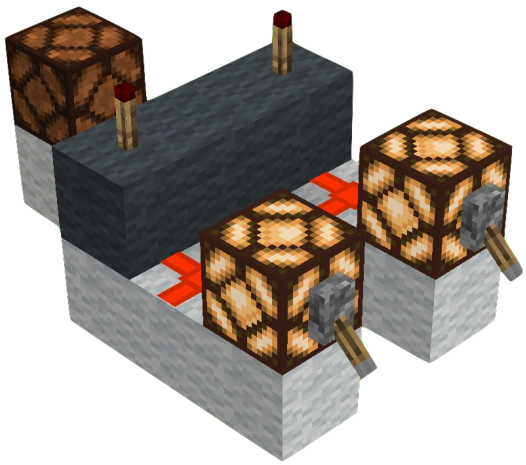

Figure: A NAND gate built in Minecraft. The output is off only when both inputs are on.

- **Circuit Diagrams:**

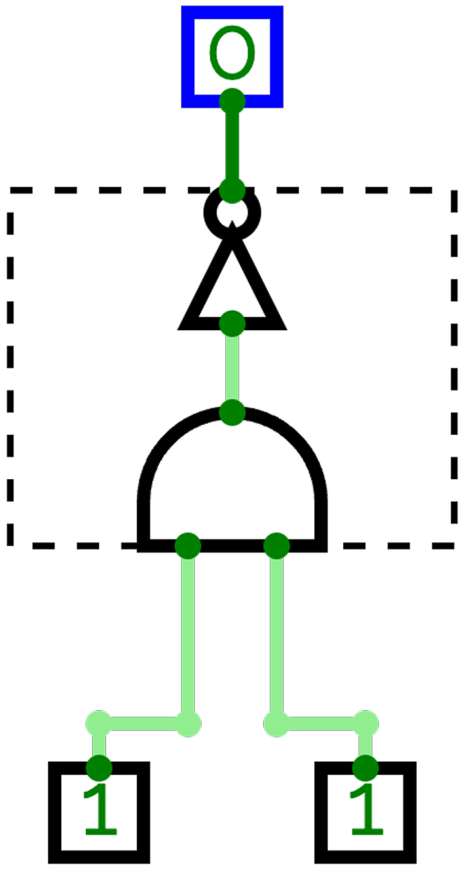

Figure: A composite NAND gate in CircuitVerse, constructed from AND and NOT gates.

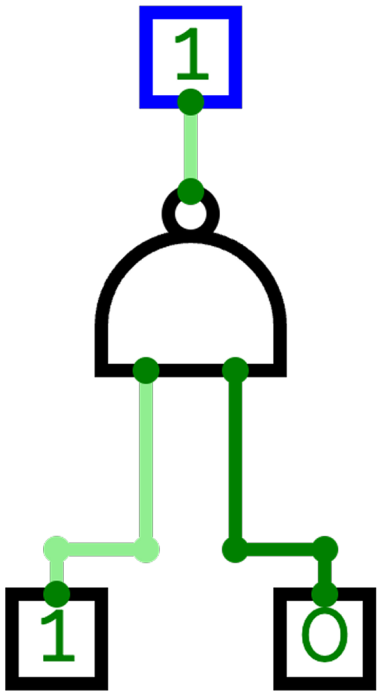

Figure: A NAND gate as shown in CircuitVerse (standard symbol).

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

    i. Build the NAND gate:

        a. Start with your AND gate setup.

b. Remove the final output torch to invert the signal.

c. Connect to an output lamp for `Y`.

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)

- **Minecraft Gate:**

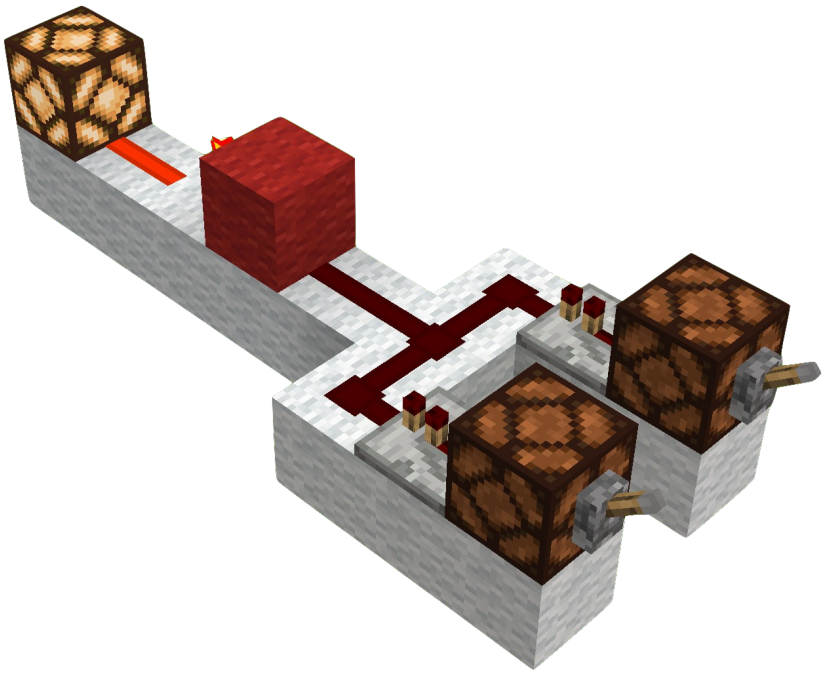

Figure: A NOR gate built in Minecraft. The output is on only when both inputs are off.

- **Circuit Diagrams:**

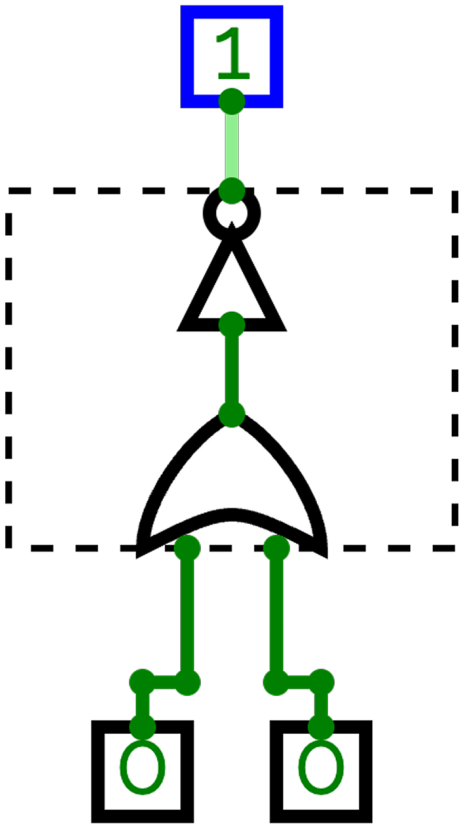

Figure: A composite NOR gate in CircuitVerse, constructed from OR and NOT gates.

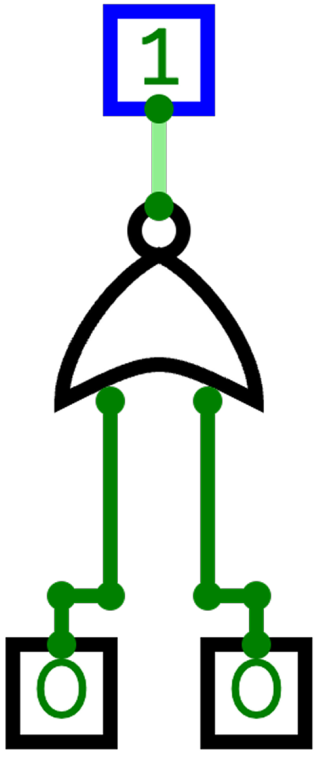

Figure: A NOR gate as shown in CircuitVerse (standard symbol).

- **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:**

  i. Build the NOR gate:

     a. Build an OR gate as before.
     b. Add a torch after the merged dust to invert the signal.
     c. Connect to an 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")

- **Minecraft Gate (Composite, Inverted Input Trick):**

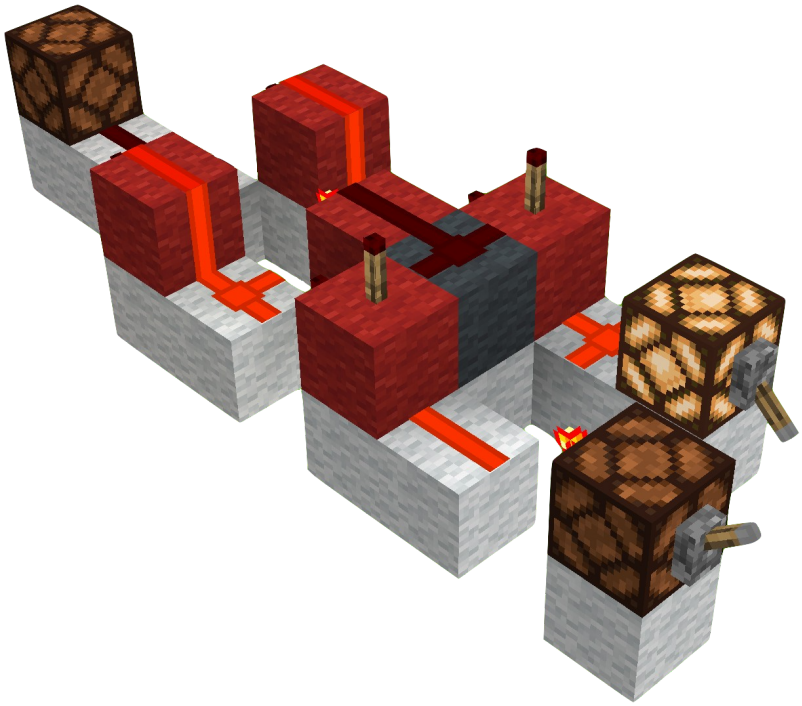

Figure: XNOR gate in Minecraft, created by inverting one input to an XOR gate. This matches the XNOR truth table: the output is on when both inputs are the same.

- **Circuit Diagram (Composite, Inverted Input Trick):**

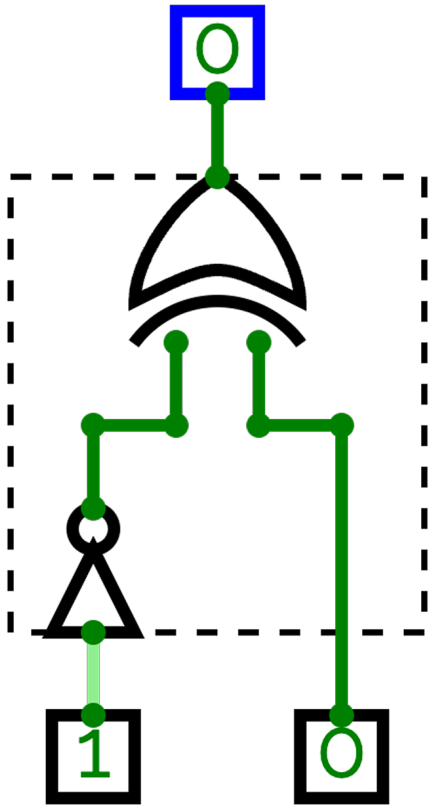

Figure: Composite XNOR gate in CircuitVerse, built by inverting one input to an XOR gate. This demonstrates that XOR(A, NOT B) is equivalent to XNOR(A, B).

- **Circuit Diagram (Standard Symbol):**

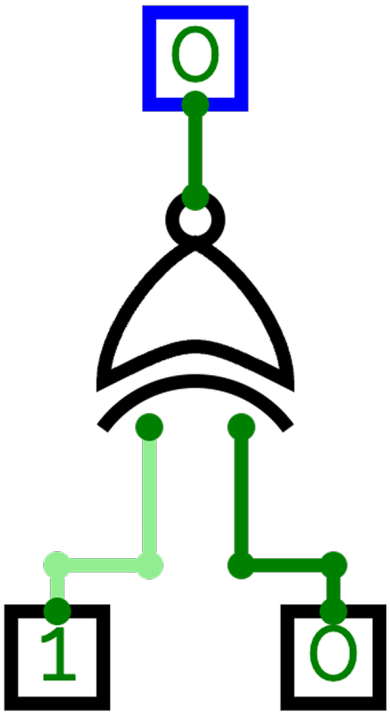

Figure: The standard XNOR gate symbol in CircuitVerse.

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

    i. Build the XNOR gate:

        a. Build an XOR gate as in Lesson 2.4.

    b. Add a torch after the output to invert it.
    c. Connect to an output lamp for `Y`.
  ii. Test all four combinations from the truth table.
  iii. **Verification:** The output is `1` when inputs match.

- **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, 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.

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.
- **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.

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! In this module, you'll build a translator that lets your computer display numbers in a way that humans can instantly recognize.

We'll break this challenge 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 register. 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`
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 register 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.