

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).

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.
-

The Journey Ahead

- **Part I: The Foundations - Speaking to the Machine.** We will begin by learning the absolute basics of Redstone and binary. We will then master the grammar of Boolean logic and use it to construct our own "keyboard" and "monitor."
 - **Module 0 (Optional):** The Redstone Toolkit
 - **Module 1:** The 4-Bit Input Interface
 - **Module 2:** The Language of Logic
 - **Interlude (Optional):** The Art of Compact Design
 - **Module 3:** Decoders & Digital Displays
 - **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 - Laying the Groundwork

Welcome to Part I of our journey to build a working computer from scratch! Every masterpiece starts with a strong foundation. In this part, we're going to master the fundamental tools, concepts, and components that will allow us (as humans) to communicate with our digital creation.

By the end of Part I, our computer won't be thinking on its own yet, but we will have a complete input and output system. You'll be able to give it a number in its native language, and it will translate that number back into a format you can instantly understand. This is where the magic begins!

Our Mission for Part I

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

- **Module 0 (Optional Prelude): The Redstone Toolkit** Learn the absolute essentials of Redstone. We'll cover the core components and the rules of power that govern every circuit we'll ever build.
- **Module 1: The 4-bit Input Interface** Build the computer's "keyboard," a simple set of levers to input numbers in binary, the machine's native language.
- **Module 2: The Language of Logic** 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.
- **Interlude (Optional): The Art of Compact Design** Bridge the gap between building for clarity and building for efficiency. A short but powerful lesson on Redstone engineering best practices.
- **Module 3: Decoders and Digital Displays** Apply your new 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 build confidence step-by-step. It's frustrating to try and build something complex if you don't first understand the tools (Module 0) or the language (Module 1). The faster you see something working, the more driven you'll be to push forward. Part I is all about giving you that tangible progress and a powerful sense of accomplishment as you complete your first fully-functional system.

Ready to start? Let's learn to handle our tools in the Redstone Toolkit

Module 0: The Redstone Toolkit – Orientation Day (Optional)

Module Summary

- Narrative Beat: Before we can speak to our computer, we need to learn how to hold the pen. This module equips you with the minimum viable skills in Minecraft's Redstone so you can confidently follow along with the rest of the course.
 - Learning Goals:
 - Identify the core components used throughout the course and understand their primary functions.
 - Grasp the fundamental concepts of Redstone power, including signal strength and Strong vs. Weak powering.
 - Build a simple "test rig" that combines the core components into a working circuit.
 - Lesson Overview:
 - Lesson 0.1: The Engineer's Toolkit
 - Lesson 0.2: How Redstone Thinks: The Rules of Power
 - Lesson 0.3: Lab: The Fundamental Circuit
 - Minecraft Artifact: A working on/off lamp circuit using a lever, wire, and output lamp.
-

Module Introduction

Welcome to Redstone University's Orientation Day!

Before we start building logic gates and registers, we need to make sure you know how to handle the tools of the trade. Think of this as our lab safety and equipment tour. Only here, the "equipment" is a mix of Redstone Dust, torches, and levers.

This is not a comprehensive Minecraft tutorial. We're here to cover only what you need for the rest of the course: the minimum viable knowledge to confidently follow along, experiment on your own, and troubleshoot when something does not work.

If you've built with Redstone before, you can likely skim this. But if you've never placed a Redstone Torch or aren't sure why a signal dies after 15 blocks, this short module will save you a lot of confusion later.

A Note on Controls & Game Setup

This course assumes you know the basic Minecraft controls for placing and breaking blocks. For an optimal learning experience, we highly recommend playing in **Creative Mode** on a **Superflat** world, which gives you unlimited resources and space to build.

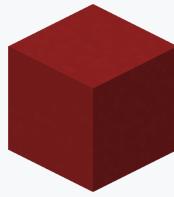
Lesson 0.1: The Engineer's Toolkit

These are the pieces you'll see over and over. They are the alphabet we will use to write our computer into existence.

Note on Texture Packs:

For clarity, I use a texture pack that enhances Redstone visibility (e.g., showing dust lines clearly). I highly recommend you find a similar one for your version of the game (such as "Vanilla Tweaks" or others). It makes debugging much easier.

Component	Game Icon	Role in this Course	Description
Redstone Dust		Wire: The foundation of all circuits.	Carries a power signal up to 15 blocks before fading. Can be placed on most solid, opaque blocks.
Redstone Torch		Power Source & Inverter (NOT Gate): Our most versatile tool.	Acts as a constant power source. When powered by another source, it turns OFF, inverting the signal. This is our primitive NOT gate .
Lever		Stable Input: Our primary way to give commands.	A simple, manual on/off switch. Perfect for setting the inputs to our computer.
Redstone Lamp		Output Indicator: Lets us see the result of a calculation.	A block that lights up when powered. We use it to visualize the state of our circuits.
Redstone Repeater		Signal Booster & Diode: Essential for complex builds.	Extends a Redstone signal back to full strength (15) and acts as a one-way diode to prevent signals from flowing backward.

Component	Game Icon	Role in this Course	Description
Solid Block		Conductor & Insulator: The physical structure of our machine.	A non-transparent block like Stone or Wool. It can be powered by Redstone components and transmit that power to adjacent components.
Sign		Documentation: A simple but vital tool for clarity.	Labeling your inputs, outputs, and different sections of a large build is crucial for understanding and debugging your own work.
Redstone Comparator		Advanced Tool (Later Modules)	We will introduce this component later when we build memory. For now, you just need to know it exists.

Lesson 0.2: How Redstone Thinks: The Rules of Power

Understanding how power travels is the single most important skill for a Redstone engineer. It can be non-intuitive, so let's establish the core rules.

Rule 1: Signal Strength & Range

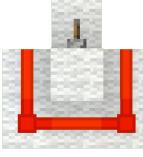
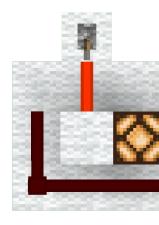
A Redstone signal has a "strength" from 15 (full power) down to 0 (off).

- A signal source (like a Lever or Torch) outputs a signal of strength **15**.
- For every block of Redstone Dust the signal travels, its strength decreases by **1**.
- After 15 blocks of dust, the signal strength is 0, and the wire goes dead.
- A **Redstone Repeater** takes any signal strength from 1 to 15 and outputs a fresh, full-strength signal of 15.

Rule 2: Strong vs. Weak Powering

This is a critical concept. Blocks can be powered in two ways, and what they can do depends on how they are powered.

	Strong Power	Weak Power
What Provides It?	A Lever, Button, Repeater, or Torch directly powering a block.	Redstone Dust running into or across a block.

	Strong Power	Weak Power
What Can It Do?	Powers all adjacent Redstone components, including dust above, below, and on all sides.	Powers only some adjacent components (like a lamp or repeater), but NOT adjacent dust.
Example		

Understanding this difference is the key to creating compact vertical circuits later in the course.

Lesson 0.3: Lab: The Fundamental Circuit

Let's combine these concepts to build a simple input to process to output circuit. This is the core pattern of every device we'll make, from simple gates to a full CPU.

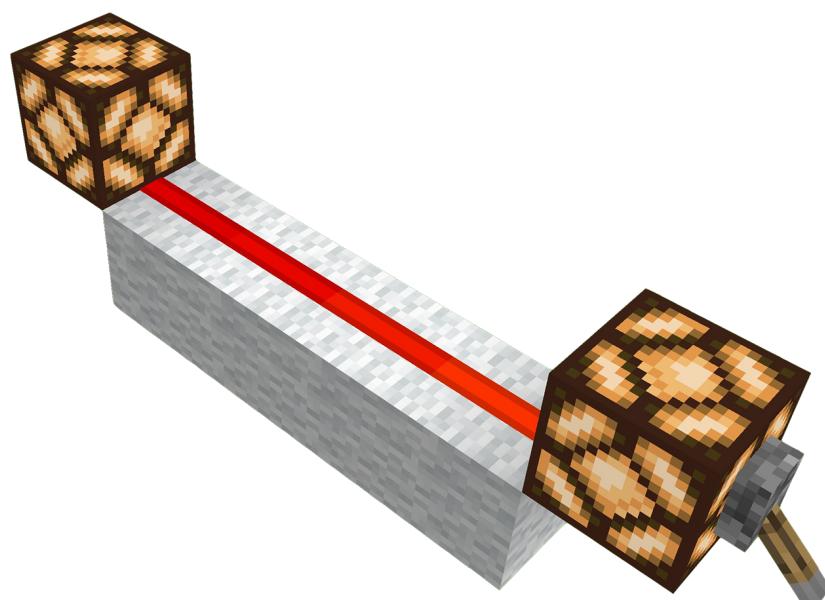


Figure: A Redstone Lamp with a lever (input) connected to a Redstone Lamp (output) through Redstone Dust (wire).

1. **Place an Output:** Place a **Redstone Lamp** on the ground.
2. **Place an Input:** A few blocks away, place a **solid block** with a **Lever** on it. You can use any solid block, but throughout the course I will use a redstone lamp with a lever as the input. This acts as a visual indicator of the input state.
3. **Wire them up:** Connect the block under the lever to the lamp using a line of **Redstone Dust**.
4. **Test:** Flip the lever. The lamp should turn on and off.
5. **Experiment:** Now, modify your circuit to test your understanding.
 - **Invert the signal:** Insert a **Redstone Torch** somewhere in the path. How does the lamp's behavior change? (Hint: The torch acts as a NOT gate).
 - **Extend the signal:** Make your Redstone Dust wire 20 blocks long. The signal will not reach. Now, place a **Repeater** after block 14. Observe how it refreshes the signal.

You've just built your first working circuit and verified the core rules of Redstone. Every single build in this course is just a more complex version of this fundamental pattern.

Module 0 Checkpoint

1. What two essential functions does a Redstone Repeater perform?
2. An engineer powers a block with a line of Redstone Dust. Will a piece of dust placed on top of that block receive power? Why or why not?
3. What Redstone component is our primitive **NOT gate**?

(Solution for this exercise is in Appendix B)

Key Terms (Module 0)

- **Signal Strength:** The power level of a Redstone signal, ranging from 15 (full) down to 0 (off). A signal loses 1 strength for every block of dust it travels.
- **Power Source:** A component, like a Redstone Torch or Lever, that outputs a full-strength (15) signal.
- **Wire:** Our term for any component, usually Redstone Dust, that transmits a signal from one point to another.
- **Input:** A component, like a Lever, that allows a user to manually control a circuit.
- **Output:** A component, like a Redstone Lamp, that displays the result or state of a circuit.
- **Strong Power:** A type of power provided by components like Repeaters or Torches directly to a block. It can activate all adjacent Redstone components, including dust.
- **Weak Power:** A type of power provided by Redstone Dust to a block. It can activate components like lamps and repeaters, but not adjacent Redstone dust.
- **Repeater:** A component that acts as a signal booster (refreshing signal strength to 15) and a diode.

- **Diode:** A component that allows a signal to flow in only one direction, preventing back-powering. The Redstone Repeater is our primary diode.
- **Inverter (NOT Gate):** A circuit or component that flips a signal from ON to OFF, or OFF to ON. The Redstone Torch is our primitive inverter.

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
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.

This input bus will serve as the starting point for our future circuits. In later modules, we'll process these binary inputs and display the results on a 7-segment display—a device that lights up segments to show numbers, like on a digital clock.

The Build Guide

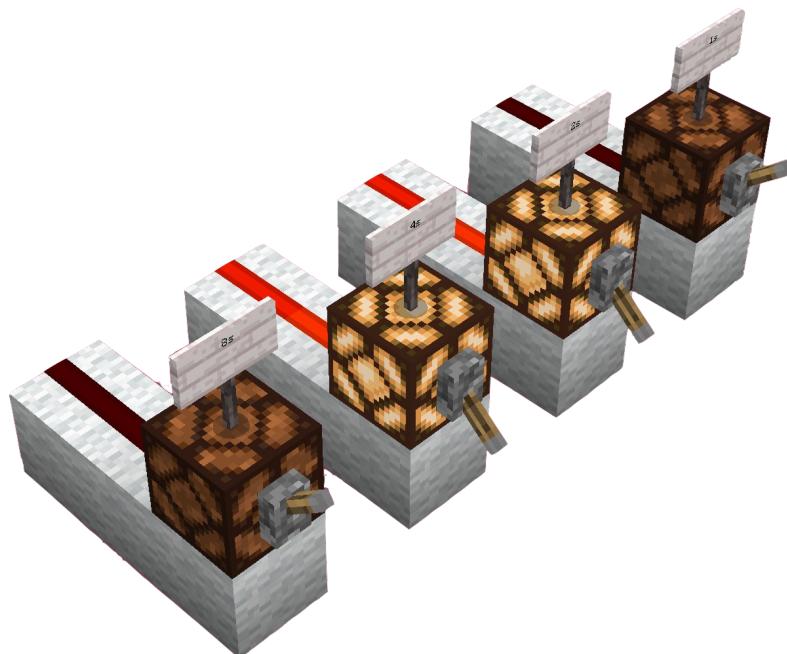


Figure: The input interface in Minecraft, set to `0110` (binary for 6). The levers are flipped to

represent the bits, and the dust is connected to the back. Using redstone lamps makes it easy to see the current state of each bit.

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

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

CircuitVerse Version

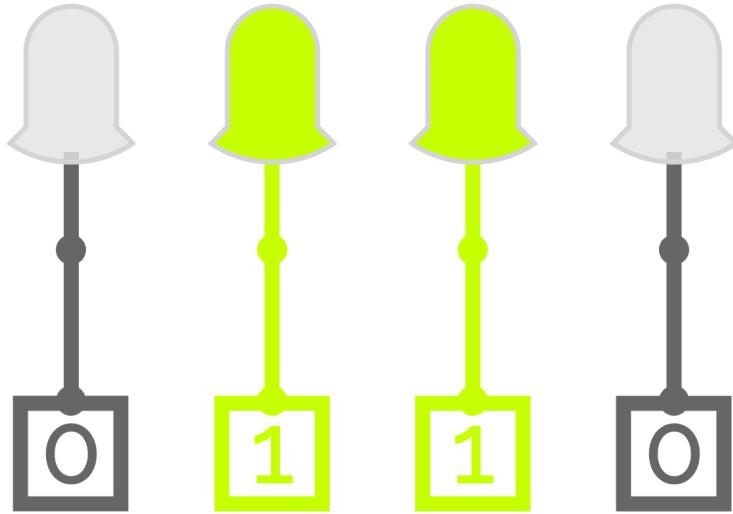


Figure: The same 4-bit input interface, built in CircuitVerse. It is also set to `0110` (6 in decimal).

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

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

Lesson 1.3: Drills & Games – Strengthening Your Binary Intuition

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

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

Drill 1: Binary to Decimal

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

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

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

The Binary "Game"

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

Lesson 1.4: Module 1 Checkpoint

Let's check our understanding before moving on.

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

Quiz

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

Real-World Connection: CPU Registers

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

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

Software Connection (LeetCode): Counting Bits

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

share.

A classic LeetCode problem is "**Number of 1 Bits**": count how many 1s 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.

Your input bus is ready to carry these binary signals to the next stage where logic gates will turn them into calculations and decisions. Now that you've built your input interface and practiced working with binary, you're ready to learn how to manipulate these binary signals using logic gates in the next module. These gates will process the inputs you've set here into meaningful outputs.

The basic building blocks of our computer are about to take shape. Get ready for the world of logic gates and circuits!

Key Terms (Module 1)

- **Binary:** A base-2 number system that uses only two symbols, 0 and 1, to represent information. It is the fundamental language of all digital computers.
- **Bit:** A single "binary digit," which can be either a 0 or a 1. It is the smallest possible unit of data in computing.
- **Decimal:** The base-10 number system that humans commonly use, with ten unique symbols (0-9).

- **Bus (Input Bus):** A collection of parallel wires that carry a complete piece of binary information. Our 4-bit input interface creates a 4-bit bus.
- **Interface (Input Interface):** A device that allows a user or system to provide information to a machine. Our 4-lever setup is a manual input interface.
- **Bitwise Operation:** An operation in software that manipulates numbers at the level of their individual bits, rather than their decimal value.
- **Register:** A small, extremely fast storage location inside a computer's central processing unit (CPU) that holds data for immediate use.

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 an input interface to send binary numbers as a signal over 4 wires forming a bus. Now we'll learn how to process that signal using Boolean algebra and logic gates.

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

To ensure a complete understanding, every logic gate is introduced using a consistent structure that moves from the abstract concept to the practical build.

- **Visual Introduction:**
 - **Abstract Symbol & Function:** We begin with an image showing the gate's standard engineering symbol alongside a simple circuit demonstrating its basic function.
 - **Composite Diagram (For Composite Gates Only):** For gates built from our primitives, we then show a detailed CircuitVerse diagram of how they are constructed using only NOT and OR gates.
 - **Minecraft Build:** Finally, we show a screenshot of the gate built in Minecraft, reflecting our "primitives-only" design philosophy.
- **Formal Definition & Rules:**
 - **Formal Definition:** The high-level concept and official terminology (e.g., "Conjunction").
 - **Symbols:** Common ways the operator is written in logic (`&`) and programming (`&&`).
 - **The Rule:** A plain-English sentence describing what the gate does.
 - **Truth Table:** A complete chart defining all possible input/output combinations. This is the ultimate "source of truth."
 - **Primitive Boolean Expression:** The specific algebraic expression that represents our composite build using only `NOT` and `OR`.

- **Practical Application:**

- **Lab & Experiment:** A hands-on test to verify your Minecraft build against the gate's 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 game mechanics 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.

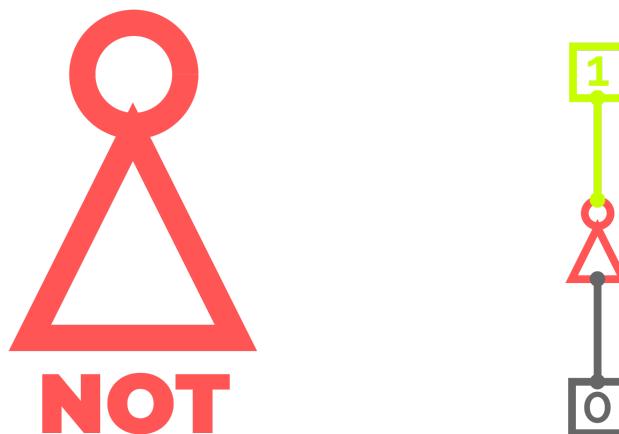


Figure: The abstract symbol for the NOT gate (left) and its function in a basic circuit (right), taking a single input A and producing an inverted output Y.

- **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:** $\neg A$ (logic), $!A$ (programming).
- **The Rule:** If the input is `True`, the output is `False`. If the input is `False`, the output is `True`.
- **Truth Table: NOT Gate**

A	$!A$
0	1
1	0

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

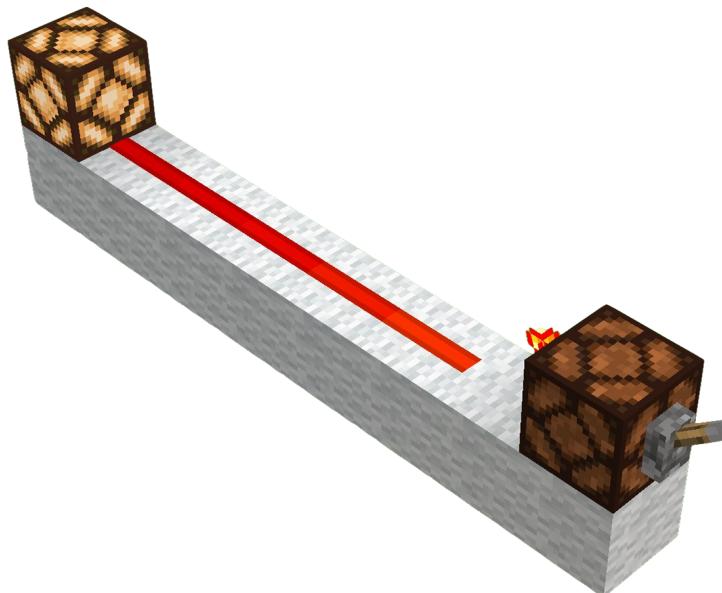


Figure: A NOT gate implemented in Minecraft using a Redstone Torch. The torch inverts the input from the lever, turning the lamp on when the lever is off and vice versa. This is the simplest physical realization of logical negation in the game.

- 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 v 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:**

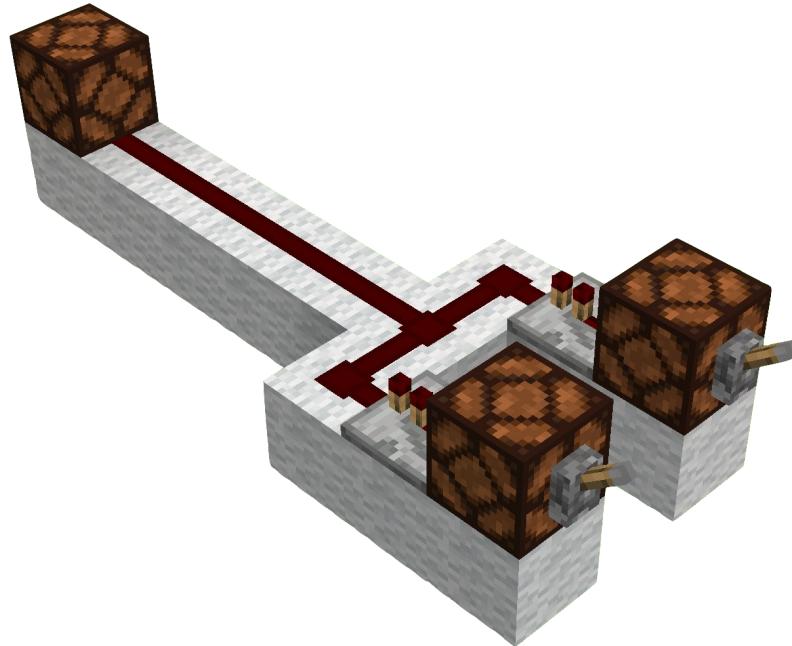


Figure: A Minecraft OR gate built by merging two Redstone Dust lines. The output lamp lights up if either lever (input A or B) is switched on, visually demonstrating the "at least one" logic of the OR operation.

- 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.

Engineering Note: What is a diode? In electronics, a **diode** is a component that allows a signal to flow in only one direction, like a one-way valve or a turnstile for electricity. This property is essential for preventing signals from going where they aren't supposed to.

In our OR gate, if we merge the dust lines directly, a signal from input A could travel backwards up the other wire and power input B's lamp, even if B's lever is off. This is called "back-powering."

The **Redstone Repeater** is a perfect, purpose-built diode in Minecraft. Notice the small arrow on top of it, it will only allow a signal to pass in that direction. By placing a repeater on each input line, we ensure the signal can flow *out* towards the final lamp, but cannot flow *backwards* to interfere with the other input.

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

iii. **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`.

Practice Problem: Boolean Expression Evaluation

Given the Boolean expression `A OR !B`, evaluate the output for all possible input combinations ($A, B = 0,0; 0,1; 1,0; 1,1$) and create a truth table. Then, build a Minecraft circuit to verify your results.

(Solution for this exercise is in Appendix B)

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.

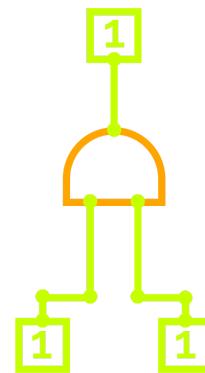
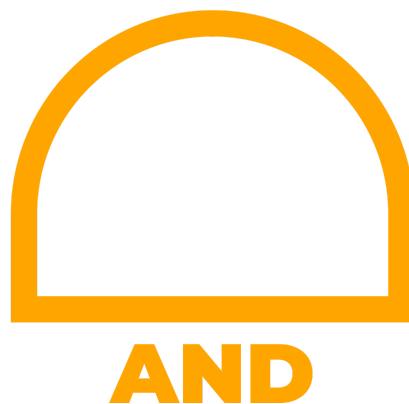


Figure: The abstract symbol for the AND gate (left) and its function in a basic circuit (right), taking two inputs A and B and producing an output Y that is active only if inputs A and B are both active.

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.

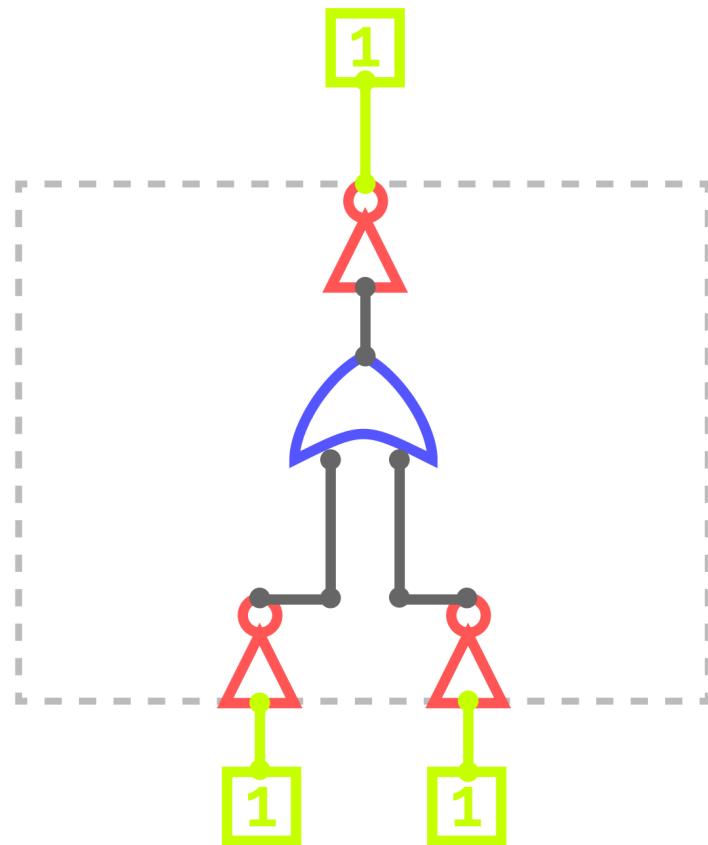


Figure: The AND gate constructed from NOT and OR gates in CircuitVerse. This composite diagram shows how two NOT gates and one OR gate are grouped to function as a single AND gate, following the logic $Y = !(A \text{ OR } B)$.

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

<code>A</code>	<code>B</code>	<code>A AND B</code>
----------------	----------------	----------------------

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

- **The Boolean Expression:** The output Y is $Y = A \text{ AND } B$. (Our build uses $!(\text{!}A \text{ OR } \text{!}B)$, 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.

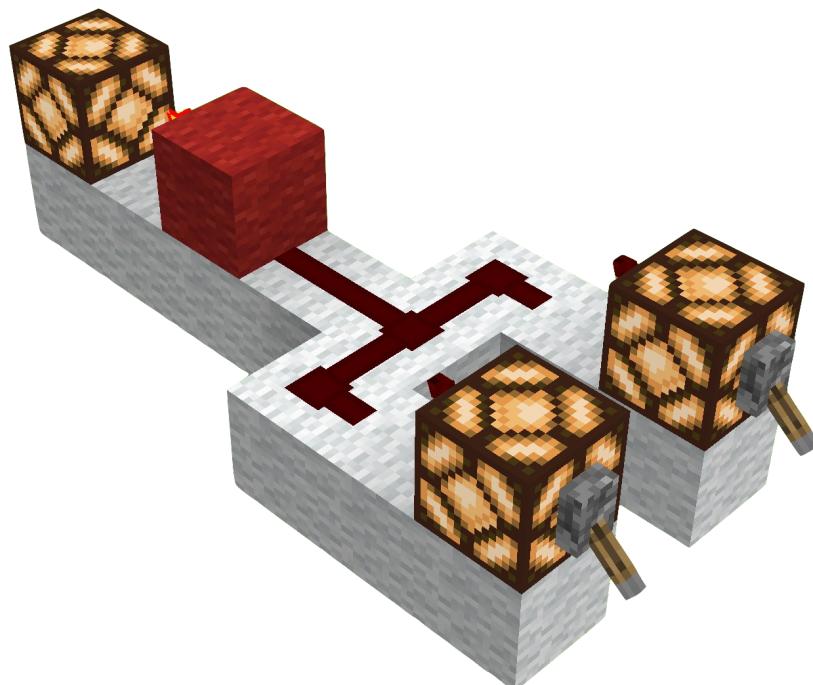


Figure: A composite AND gate in Minecraft, constructed using two Redstone Torches (NOT

gates) and a Redstone Dust merger (OR gate), then inverted again. This build demonstrates how to achieve AND logic using only the game's primitive components.

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. Place 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 .

Practice Problem: Logic Gate Design Challenge

Design a circuit that implements the logic A AND !B using only the NOT and OR primitives (no direct AND gate). Build it in Minecraft and verify with a truth table.

(Solution for this exercise is in Appendix B)

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 AB), **OR** as addition (A + B), and **NOT** as an overbar (\bar{A}).

For this course, we will use a specific convention designed for maximum clarity:

- We will use the words **AND** and **OR** in our expressions, as they are unambiguous.
- For **negation**, we will use the exclamation mark (**!**), as in **!A**. This is the standard symbol used in many programming languages and keeps our complex expressions clean and readable.

The Laws of Boolean Algebra

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

- **Identity Law:** $A \text{ OR } 0 = A$ and $A \text{ AND } 1 = A$.
 - **Annihilator Law:** $A \text{ OR } 1 = 1$ and $A \text{ AND } 0 = 0$.
 - **De Morgan's Law:** This is the superstar. It gives us a way to convert between ANDs and ORs.
 - $!(A \text{ AND } B)$ is the same as $!A \text{ OR } !B$
 - $!(A \text{ OR } B)$ is the same as $!A \text{ AND } !B$
-

Lab 1: Proving a Circuit with De Morgan's Law

Let's use De Morgan's Law to prove our AND gate design is correct.

1. The two redstone torches on the back of our redstone lamps are NOT gates, giving us **!A** and **!B**.
 2. Their signals merge into the central spot, which is an OR gate (**!A OR !B**).
 3. The final output torch is a NOT gate on that signal. Therefore, the full expression for our circuit is **!(!A OR !B)**.
 4. Applying De Morgan's Law to the part in the parentheses: **!A OR !B** is the same as **!(A AND B)**.
 5. Substituting that back in, our expression becomes **!(!(A AND B))**.
 6. The two NOTs (**!!**) cancel each other out, leaving **A AND B**. We just proved our physical circuit is correct!
-

Lab 2: Proving a Circuit with the Distributive Law

The laws of logic don't just prove that a circuit is correct; they can also make our circuits much simpler. This is a crucial engineering skill called **simplification**.

Consider a circuit that needs to turn on if **(A is ON and B is ON)** OR if **(A is ON and B is OFF)**. The direct Boolean expression would be:

$$Y = (A \text{ AND } B) \text{ OR } (A \text{ AND } !B)$$

This looks like it would require two AND gates and one OR gate. Let's use the laws of logic to simplify it.

1. **Start with the expression:** $Y = (A \text{ AND } B) \text{ OR } (A \text{ AND } !B)$
2. **Apply the Distributive Law:** Notice that **A AND** is common to both terms. We can "factor it out."
 - This gives us: $Y = A \text{ AND } (B \text{ OR } !B)$

3. Apply the Inverse Law: We know that an input OR'd with its own inverse ($B \text{ OR } !B$) is always equal to 1 (True).

- The expression becomes: $Y = A \text{ AND } 1$

4. Apply the Identity Law: We know that any input AND'd with 1 is just itself.

- The final expression is: $Y = A$

Lab Takeaway: We have just proven that this entire three-gate circuit can be replaced by a single wire connected to input A . This is the power of simplification in action. It saves resources, space, and makes our designs more elegant.

Summary Table: Boolean Laws

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

A Special Note on the Distributive Law: Notice that Boolean algebra has two distributive laws. The first one, $A \text{ AND } (B \text{ OR } C)$, looks very similar to the distributive law in regular algebra. However, the second one, $A \text{ OR } (B \text{ AND } C)$, is unique to Boolean logic. In the algebra you're used to, $a + (b * c)$ does NOT equal $(a + b) * (a + c)$. This unique property of duality is one of the things that makes Boolean algebra so powerful for simplifying digital circuits.

Functional Completeness: Building with Universal Gates

Universal Gate	To Build a NOT Gate ($\neg A$)	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.

Practice Problem: Circuit Simplification Challenge

Given the expression $(A \text{ OR } B) \text{ AND } (\neg A \text{ OR } \neg B)$, simplify it using Boolean laws.

(Solution for this exercise is in Appendix B)

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.

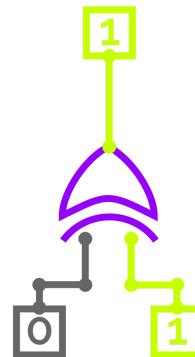


Figure: The abstract symbol for the Exclusive OR (XOR) gate (left) and its function in a basic circuit (right), taking two inputs A and B and producing an output Y that is active only if the inputs are different.

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

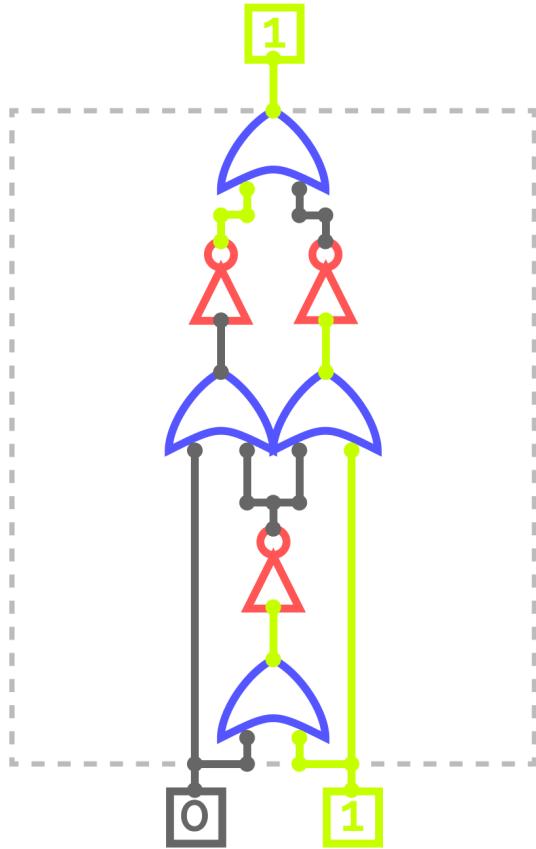


Figure: The XOR gate constructed in CircuitVerse using only OR and NOT gates. This composite design highlights how the XOR function can be achieved by creatively wiring together these basic primitives.

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

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

A	B	$A \oplus B$
---	---	--------------

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

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

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, built by combining Redstone Dust (OR logic) and Redstone Torches (NOT logic). The output lamp lights only when the two input levers are set to different states, illustrating the exclusive nature of XOR.

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).

Practice Problem: Two-Switch Light System

Design a Minecraft circuit for a two-switch light system where flipping either switch toggles the light's state (on to off, or off to on). Use only NOT and OR gates to implement the XOR logic.

(Solution for this exercise is in Appendix B)

Lesson 2.5: Software Superpowers – The XOR Trick for Programmers

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

Why is XOR so useful in programming?

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

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

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

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

Example Problem: The "Single Number"

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

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

Your Turn: The "Missing Number" Challenge

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

The Challenge:

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

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

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

find the single difference between these two groups?

(Solution for this exercise is in Appendix B)

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: NOR (The "Neither" Gate)



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.

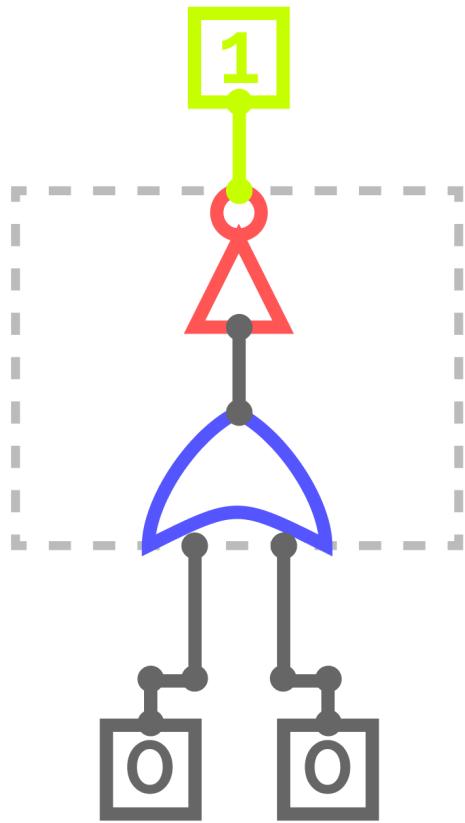


Figure: A composite NOR gate in CircuitVerse, constructed using only OR and NOT gates. The output is high only when both inputs are low, demonstrating NOR logic using our primitive gates.

- **Formal Definition:** The NOR gate performs a **NOT-OR** operation (negation of OR).
- **Symbols:** `A NOR B` or $\neg(A \vee B)$.
- **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 $Y = !(A \text{ OR } B)$.
- **Lab & Experiment:**

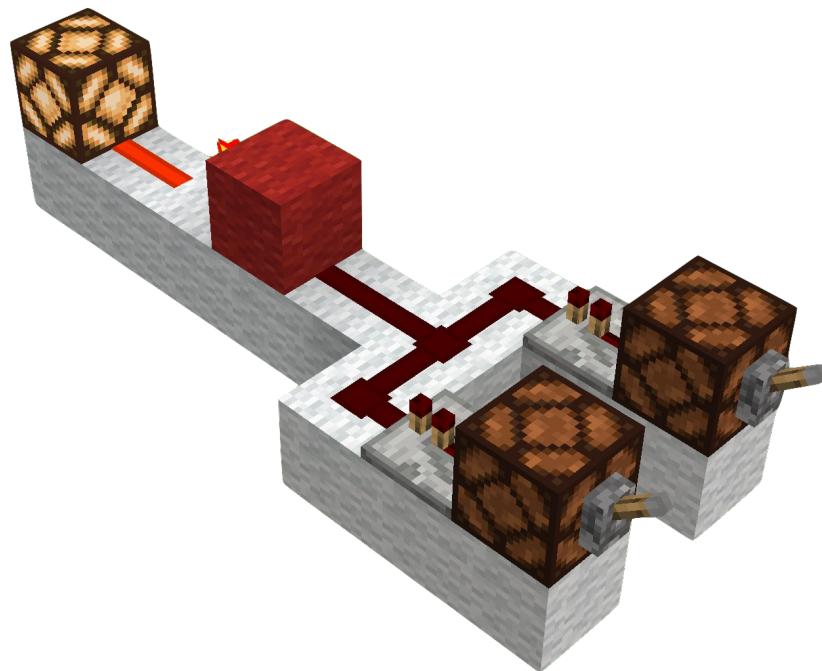


Figure: A NOR gate in Minecraft, created by merging two Redstone Dust lines (OR logic) and then inverting the result with a Redstone Torch. The output lamp lights up only when both input levers are off, demonstrating the NOR operation.

i. Build the NOR gate:

- Build the OR gate as in Lesson 2.2.
- Between the merged dust and the output lamp, place a block with a torch on the output side to invert the signal.
- 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 5: NAND (The "Not Both" Gate)



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.

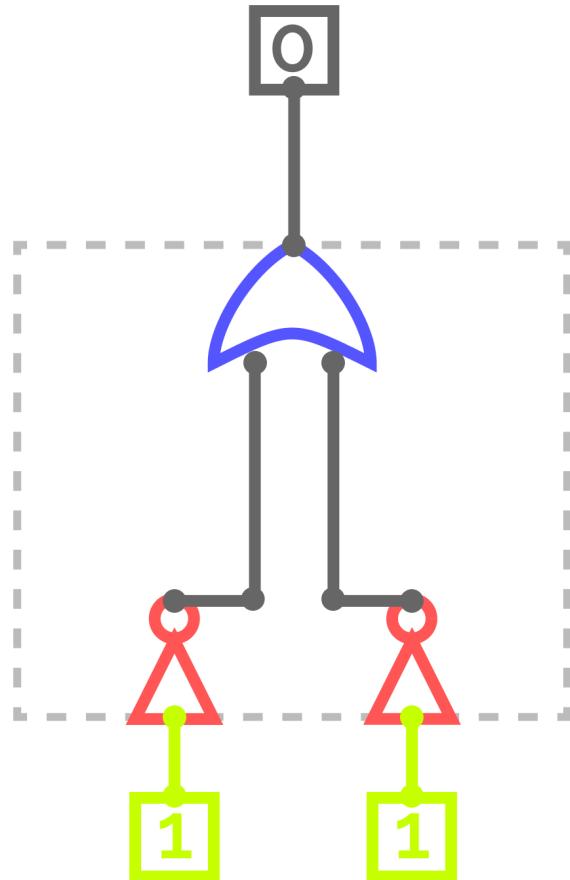


Figure: A composite NAND gate in CircuitVerse, constructed using only OR and NOT gates. This diagram shows how the NAND function can be achieved by inverting the output of a composite AND gate built from these primitives, without using a dedicated AND gate block.

- **Formal Definition:** The NAND gate performs a **NOT-AND** operation (negation of AND).

- **Symbols:** A NAND B or $\neg(A \wedge B)$.
- **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:** $Y = !A \text{ OR } !B$

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 !(A AND B). We also know from De Morgan's Law that !(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 ! (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:**

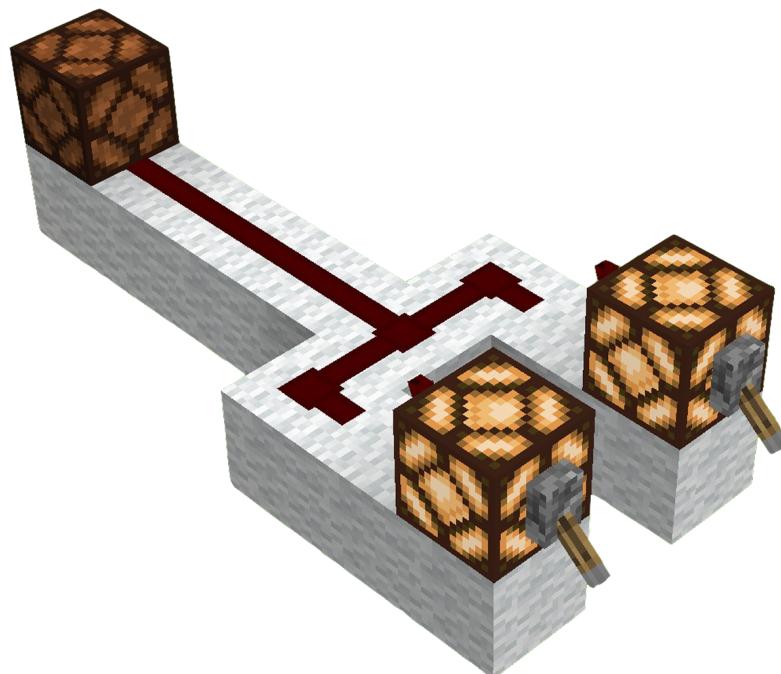


Figure: A NAND gate in Minecraft, constructed by modifying the composite AND gate and tapping the output before the final inversion. The output lamp turns off only when both input levers are on, matching the NAND truth table.

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 6: XNOR (The "Equality Detector")

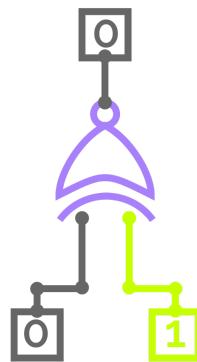
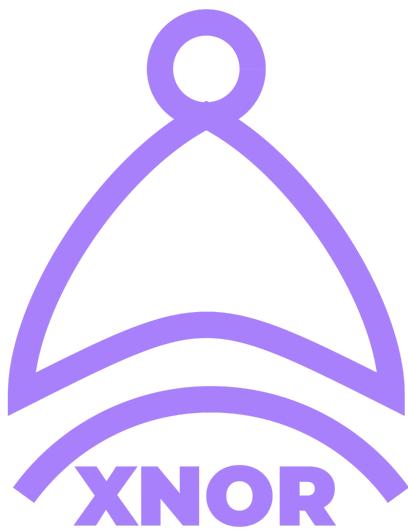


Figure: The abstract symbol for the Exclusive NOR (XNOR) gate (left) and its function in a basic circuit (right), taking two inputs A and B and producing an output Y that is active only if the inputs are the same.

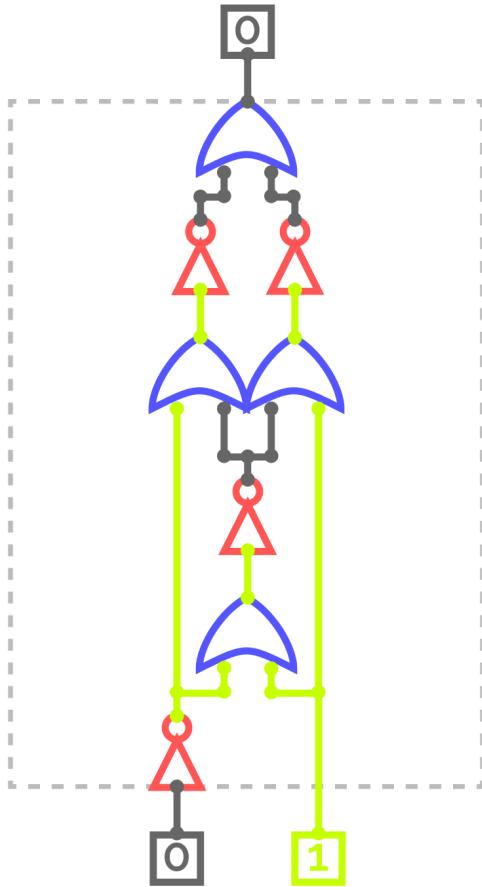


Figure: Composite XNOR gate in CircuitVerse, constructed using only OR and NOT gates. This diagram shows how XNOR logic can be achieved by inverting one input to a composite XOR gate built from these primitives, demonstrating the equivalence between $\text{XOR}(A, \text{NOT } B)$ and $\text{XNOR}(A, B)$.

- **Formal Definition:** The XNOR gate performs a **NOT-XOR** operation (negation of XOR).
- **Symbols:** `A XNOR B` or `$\neg(A \oplus B)$` .
- **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 = !(!(A OR !(A OR B)) OR !(B OR !(A OR B)))`
- **Lab & Experiment:**

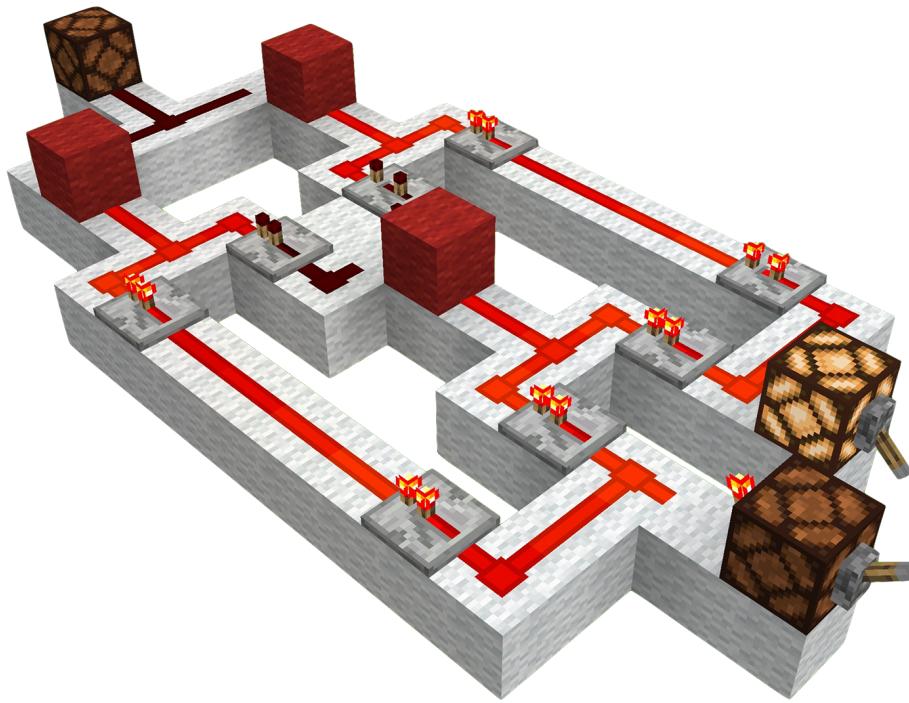


Figure: An XNOR gate in Minecraft, constructed by adding a NOT gate to one input of a composite XOR gate. The output lamp lights up only when both input levers are set to the same state, visually confirming the XNOR truth table.

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 the **logical negation** of our entire complex XOR expression
- 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.

Practice Problem: Universal Gate Challenge with NOR Gates

Build an AND gate using only NOR gates. Verify it with a truth table in Minecraft.

(Solution for this exercise is in Appendix B)

Lesson 2.7: Module 2 Summary

You have reached the end of the learning portion of this module. You started with the basic idea of True and False and have now built a complete toolkit of the seven fundamental logic gates. You've learned how to describe them with truth tables and Boolean expressions, how to build them from our primitives, and how they connect to both real-world hardware and software.

This summary table provides a single place to review all the gates at a glance.

Logic Gates Summary Table

Gate	Symbol	Core Logic Rule	Primitive Boolean Expression
NOT		Inverts a single input.	!A
OR		True if at least one input is True.	A OR B
AND		True only if all inputs are True.	$\text{!(\text{!A OR !B})}$

Gate	Symbol	Core Logic Rule	Primitive Boolean Expression
XOR		True only if inputs are different .	$!(A \text{ OR } !(A \text{ OR } B)) \text{ OR } !(B \text{ OR } !(A \text{ OR } B))$
NAND		True unless all inputs are True.	$!A \text{ OR } !B$
NOR		True only if all inputs are False.	$!(A \text{ OR } B)$
XNOR		True only if inputs are the same .	$!(!(A \text{ OR } !(A \text{ OR } B)) \text{ OR } !(B \text{ OR } !(A \text{ OR } B)))$

Lesson 2.8: Module 2 Checkpoint

Module Summary: You have reached the end of the most theory-intensive module in this course. You began with simple on/off switches and have now mastered the seven fundamental logic gates, the Boolean laws that govern them, the power of simplification, and the bridge between hardware logic and software problem-solving. It is time to test your newfound knowledge.

This checkpoint is divided into three parts to test the different skills you've acquired:

- **Part 1: Knowledge Check** - Quick questions to test your memory and understanding of core concepts.
- **Part 2: Logic Puzzles** - "On-paper" challenges requiring you to apply the laws of Boolean algebra.
- **Part 3: The Debug Challenge** - A practical, in-game challenge to test your troubleshooting skills.

Part 1: Knowledge Check

Test your core understanding with these rapid-fire questions.

1. What is the key difference in the output of an OR gate versus an XOR gate?
2. Which two gates are considered "universal," and what is the name of this property?
3. Using De Morgan's Law, what is the equivalent expression for $!(A \text{ OR } B)$?

(Solution for this exercise is in Appendix B)

Part 2: Logic Puzzles

Apply the laws of Boolean algebra to solve these challenges on paper.

Puzzle 1: The Word Problem

A greenhouse has an automated climate control system. An alarm Y should sound if the following conditions are met:

- The system is in "Manual Override" mode (M is True), OR
- The Temperature T is too high AND the Water Sprinklers W have failed to turn on (W is False).

Write the single Boolean expression for the alarm Y .

(Solution for this exercise is in Appendix B)

Puzzle 2: The Simplification

An engineer has designed a circuit with the expression: $Y = (A \text{ AND } B) \text{ OR } (A \text{ AND } !B) \text{ OR } (!A \text{ AND } B)$.

This seems to require three AND gates and two OR gates. Simplify this expression to its most efficient form using Boolean laws.

(Solution for this exercise is in Appendix B)

Part 3: The Debug Challenge (In-Game)

In the world download for this module, you will find a section labeled "Module 2 Debug Challenge." I have built a circuit that is *supposed* to implement the logic for the greenhouse alarm from Part 2: $M \text{ OR } (T \text{ AND } !W)$.

However, it's giving the wrong output for some input combinations! Your mission is to use your knowledge of truth tables and circuit tracing to diagnose the mistake in the Redstone wiring and fix it so it functions correctly. Good luck!

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 apply these logic gates to build our first truly complex and useful machine: a translator that will convert binary inputs into signals for a 7-segment display, allowing our computer to show numbers in a way that’s easy for humans to read.

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.
- **Diode:** A component that allows an electrical signal to pass in only one direction. In Minecraft, the Redstone Repeater acts as a perfect diode.
- **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 (B): The Art of Compact Design (Optional)

note: This is one potential version of this interlude. I haven't decided which approach to take yet.

A Note from the Instructor

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

Before we begin our next major project, we have this special, optional section. Think of it as an engineering deep dive. The goal of Module 2 was to build for **clarity**, making our gates large so the logic was easy to trace. This Interlude introduces the art of building for **efficiency**.

We will analyze some common, space-saving designs used by the Redstone community. Understanding them is not required for the rest of the course, but it will empower you to make your own builds smaller and faster. This is your first step from being a student of logic to becoming a true Redstone engineer.

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

Every engineering decision is a compromise. When you compact a circuit, you are usually trading **readability** for **efficiency**.

Factor	Verbose (Educational) Builds	Compact (Practical) Builds
Size / Footprint	Large and sprawling for clarity.	Small and dense to save space in large machines.
Speed / Tick Delay	Often slightly slower due to longer wire paths.	Can be faster with shorter signal paths.
Readability	Very easy for a human to trace and debug.	Can be cryptic and difficult to troubleshoot.

Guideline: For learning and debugging, verbose is best. For final builds where space and resources matter, compact is essential.

Case Studies in Compact Design

Let's analyze a few classic compact designs. For each one, we'll compare the **Verbose Teaching Version** you already built with a **Compact Practical Version** and break down how it works.

Case Study 1: The AND Gate

First, recall our verbose AND gate. It's a perfect physical representation of De Morgan's Law, $\neg(\neg A \text{ OR } \neg B)$, but it takes up a lot of room.

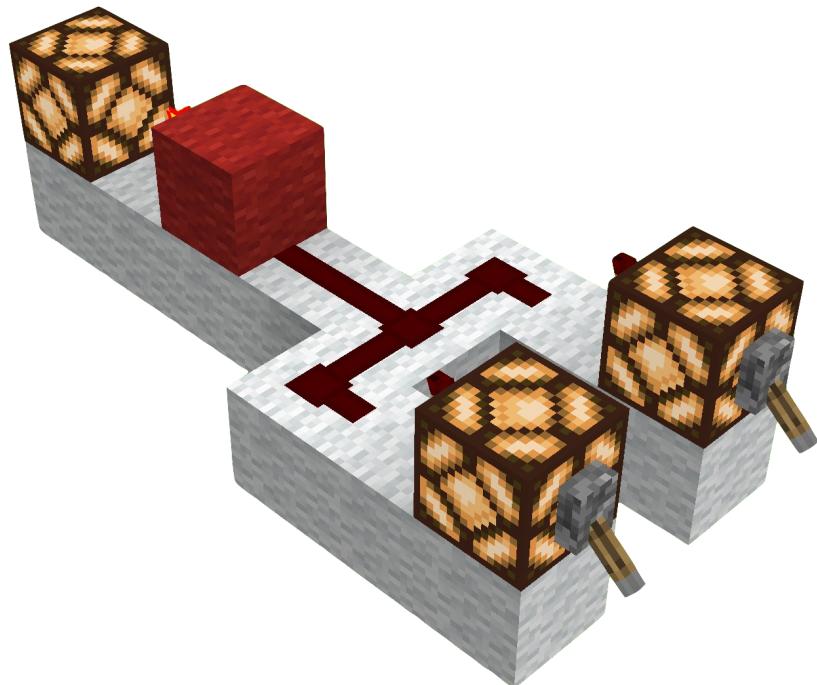


Figure: Our easy-to-read, but large, educational AND gate.

Now, observe a classic compact AND gate. It performs the exact same function in a tiny 3x2 footprint.

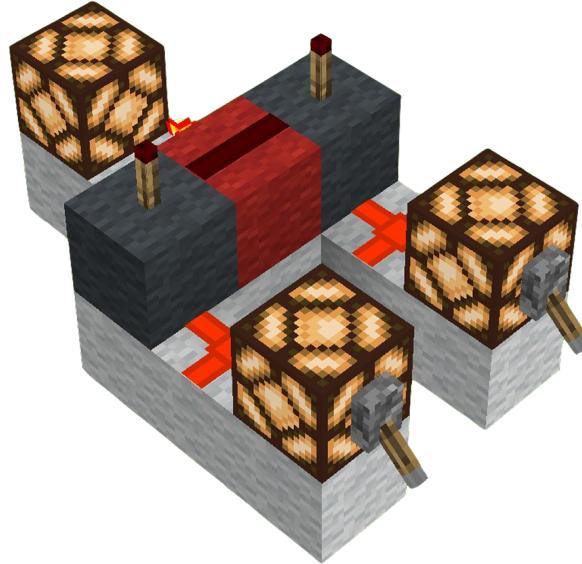


Figure: A classic, space-efficient compact AND gate.

Logical Deconstruction: This compact build is a brilliant physical implementation of the same logic.

- The two torches on the sides of the input blocks are your first **NOT** gates, creating $\neg A$ and $\neg B$.
- The central Redstone dust is the **OR** gate. It gets powered if *either* of the side torches turns off.
- The torch on the front of the central block is the final **NOT** gate, inverting the signal from the dust. The logic is identical: $\neg(\neg A \text{ OR } \neg B)$. It's just cleverly folded into a smaller space by using how torches and dust interact.

Case Study 2: The XOR Gate

Our educational XOR gate is large because the logic is complex. It's designed to be read.

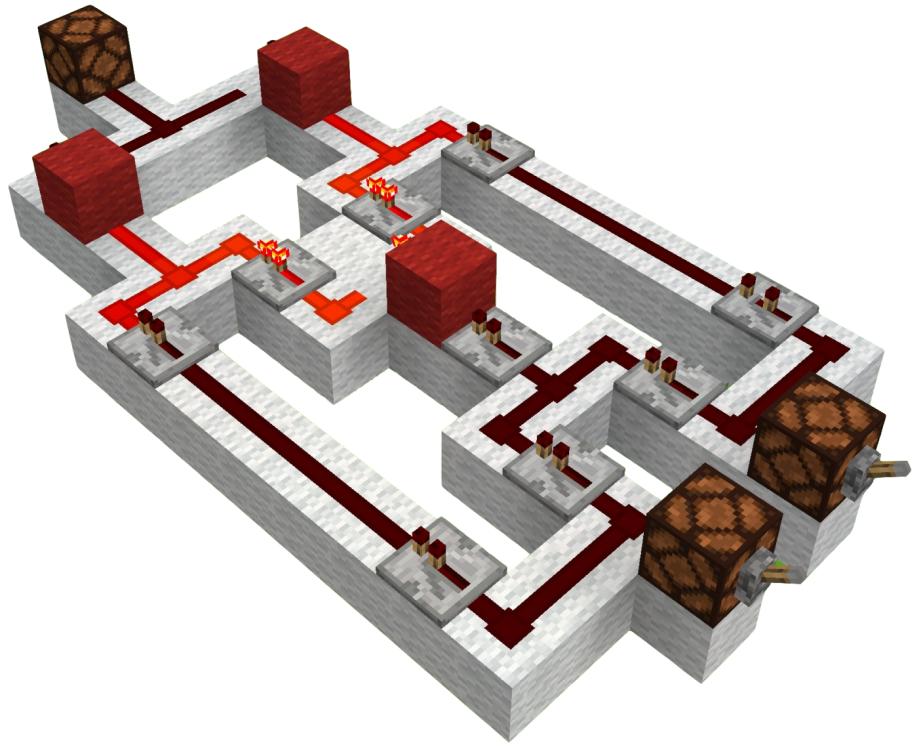


Figure: Our educational XOR gate, built for clarity.

The community has created many compact XOR designs. Here is one of the most common "tileable" (meaning you can place them side-by-side) versions.

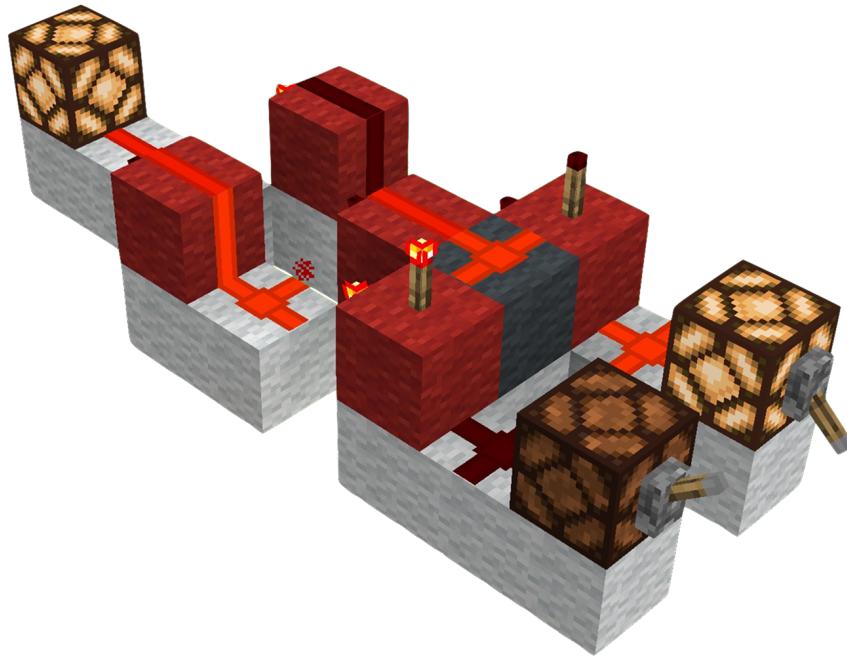


Figure: A very common and tileable compact XOR gate design.

Logical Deconstruction: This design is a masterclass in efficiency. It cleverly uses torch burnout and block power states to create the two conditions for an XOR ($A \text{ AND } !B$ or $!A \text{ AND } B$) and merges their outputs. While tracing the exact path is advanced, the key takeaway is that it perfectly matches the XOR truth table in a minimal amount of space, which is critical when you need to build dozens of them for an arithmetic unit.

Conclusion: Your Journey Into Optimization

You now see the difference between a circuit designed for teaching and one designed for a practical machine. Compact designs aren't magic; they are just clever physical implementations of the same Boolean logic you have already mastered.

From Module 3 onward, we will follow the **Rule of Abstraction**:

A logic gate is defined by its **truth table** (its inputs and outputs), not by its internal layout. You are now free to use the verbose educational builds, the compact practical builds, or any other design that functions correctly.

This freedom is a major step in your journey from student to engineer.

Explore More: The Gate Museum

In the world download provided for the course, you will find a section labeled "Gate Museum" which showcases these and many other community-tested compact designs for each logic gate. I encourage you to explore, build, and test them to expand your engineering toolkit.

Interlude I (A): The Art of Compact Design (Optional)

note: This is one potential version of this interlude. I haven't decided which approach to take yet.

A Note from the Instructor

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

Before we begin our next major project, we have this special, optional section. Think of it as an engineering deep dive. The goal of Module 2 was to build for **clarity**, making our gates large so the logic was easy to trace. This Interlude introduces the art of building for **efficiency**.

We will analyze some common, space-saving designs used by the Redstone community. Understanding them is not required, but it will empower you to make your own builds smaller and faster. This is your first step into true Redstone engineering. From Module 3 onward, we will follow the **Rule of Abstraction**: a gate is defined by its function, not its layout. You are free to use these new designs or the verbose ones from Module 2. As long as it works, it is correct.

Interlude Summary

- Narrative Beat: You have 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 classic compact gate designs to see these principles in action.
 - Minecraft Artifact: A toolkit of compact logic gate designs.
-

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

Every Redstone design is a compromise. When you compact a circuit, you are usually trading **readability** for **efficiency**.

Factor	Verbose (Educational) Builds	Compact (Practical) Builds
Size / Footprint	Large and sprawling for clarity.	Small and dense to save space.

Factor	Verbose (Educational) Builds	Compact (Practical) Builds
Speed / Tick Delay	Often slower due to longer wiring.	Can be faster with shorter signal paths.
Readability	Very easy to trace and debug.	Can be difficult to read, making errors harder to find.

Guideline:

For learning and debugging, verbose is best. For large-scale builds where space is a concern, compact is essential.

Case Studies in Compact Design

Each case study compares the **Verbose Teaching Version** you learned in Module 2 with a **Compact Practical Version** designed for efficiency. Both obey the same truth table, the only difference is the implementation.

Case Study: The Compact AND Gate

First, recall our verbose AND gate, built from our primitives to be easy to understand.

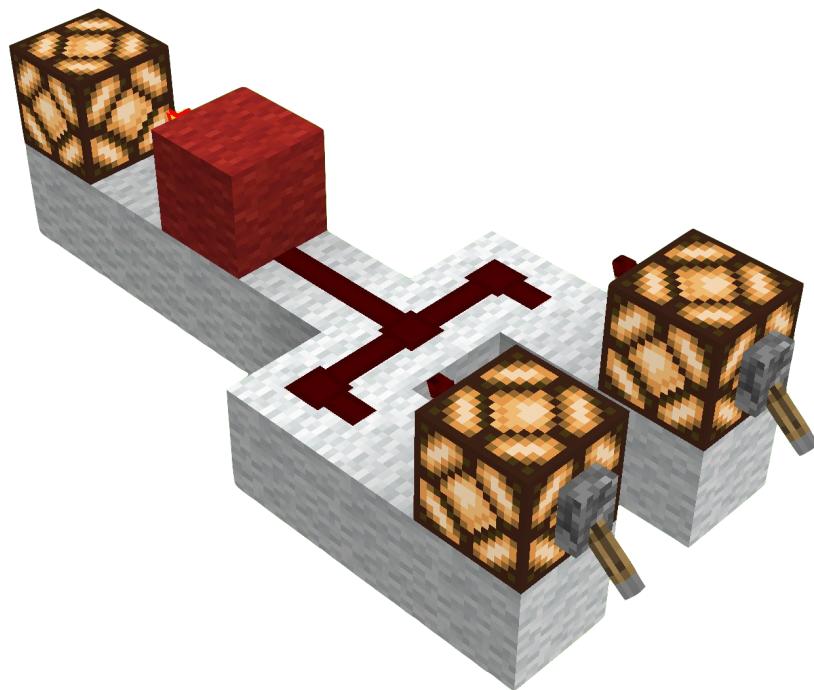


Figure: Our easy-to-read, but large, educational AND gate.

Now, look at the design below. It performs the exact same logic, but in a much smaller space.

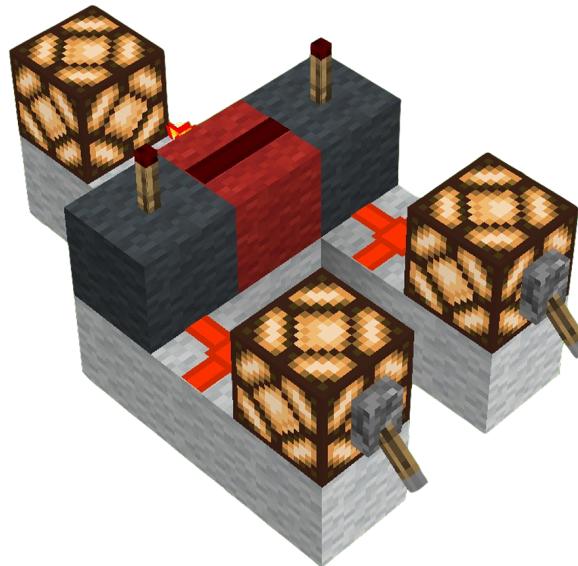


Figure: A classic, space-efficient compact AND gate.

Logical Deconstruction

This compact build still follows the logic of $!(\text{!}A \text{ OR } \text{!}B)$. The components are just cleverly merged.

- The two torches on the sides are the first `NOT` gates, creating `!A` and `!B`.
- The central Redstone dust on top of the block acts as the `OR` gate, collecting the signals from the torches. If `!A` is on, OR `!B` is on, this dust becomes powered.
- The torch on the back of the central block is the final `NOT` gate, inverting the `OR` signal to produce the final AND output.

Build Steps

1. Place three solid blocks in a row.
2. Place input levers on the front of the two outer blocks.
3. Place a Redstone Torch on the top of each of the two outer blocks. These are your `!A` and `!B` inverters.

4. Place a single piece of Redstone Dust on top of the central block. This dust will be powered by either of the torches. This is your `OR` gate.
 5. Place a Redstone Torch on the back face of the central block. This is your final inverter.
 6. The output from this final torch is `A AND B`. Connect it to a lamp to test.
-

Case Study: The Compact NAND Gate



Verbose NAND Gate in Minecraft

Figure: Our educational NAND gate, built by removing the final inverter from the verbose AND gate.



Compact NAND Gate in Minecraft

Figure: A compact NAND gate, created by tapping the output of the compact AND gate before the final inversion.

Logical Deconstruction

This is the beauty of understanding the logic. A NAND gate is simply $!(A \text{ AND } B)$. In our compact AND gate build, the final component is a torch that performs the last inversion. To get a NAND output, we just need to take the signal from *before* that final torch. The signal on the central dust is `!A OR !B`, which De Morgan's Law proves is logically identical to `A NAND B`.

Build Steps

1. Build the compact AND gate exactly as described above.
 2. Instead of taking the output from the final torch, connect your output wire directly to the Redstone Dust on top of the central block.
 3. This signal is your NAND output. Connect it to a lamp to verify.
-

Case Study: The Compact XOR Gate

Recall our large, easy-to-read XOR gate from Module 2.



Figure: Our educational XOR gate, built for clarity.

Now, observe a classic compact XOR design. It is much smaller but harder to read at a glance.

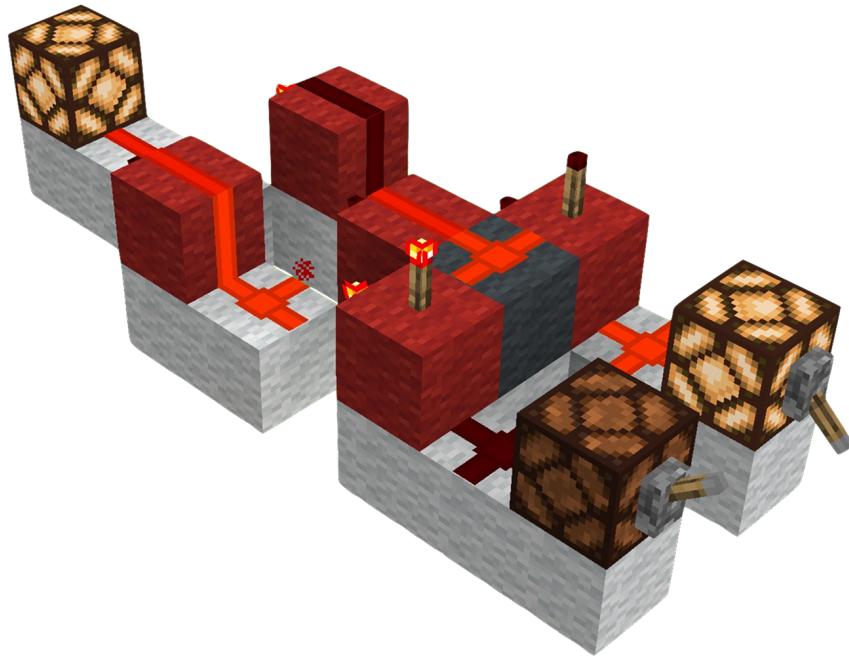


Figure: A very common and tileable compact XOR gate design.

Logical Deconstruction

This design is more complex and relies on how torches interact. The logic effectively creates the two conditions for an XOR ($A \text{ AND } !B$ and $!A \text{ AND } B$) in parallel and merges their outputs.

- The torches on top of the input blocks create $!A$ and $!B$.
- The torches on the sides of the output block are each controlled by a combination of a direct input and an inverted input.
- The final Redstone dust on top acts as an OR gate, combining the outputs of the side torches.

Build Steps

1. Place two solid blocks for your inputs, side-by-side. Place levers on the front.
2. Place a third solid block one block away from and between the inputs. This will be your output block.
3. On the input blocks, place Redstone Dust on top.
4. On the output block, place Redstone Torches on the two sides facing the input blocks.
5. On top of the output block, place a piece of Redstone Dust.
6. Place Redstone Torches on the faces of the input blocks that face the central output block.
7. The final output is taken from the Redstone Dust on top of the output block. Connect it to a lamp. It will be on only when the inputs differ.

Case Study: The Compact XNOR Gate



Verbose XNOR Gate in Minecraft

Figure: The educational XNOR, built by inverting an input to the verbose XOR gate.



Compact XNOR Gate in Minecraft

Figure: A compact XNOR gate, built by simply inverting one input to the compact XOR gate.

Logical Deconstruction

Just as we learned in Module 2, the simplest way to create an XNOR (equality detector) is to invert one of the inputs to an XOR gate. The logic $A \text{ XNOR } B$ is identical to $A \text{ XOR } !B$. We can apply this exact same principle to our compact build. This is the art of the engineer: modifying an existing component instead of reinventing the wheel.

Build Steps

1. Build the compact XOR gate exactly as described above.
 2. Choose one input, for example, input B.
 3. To invert it, place a Redstone Torch between the lever for B and the input block of the XOR gate.
 4. Now, flipping the B lever will send an inverted signal into the compact XOR circuit. The entire circuit will now behave as an XNOR gate, lighting up only when the two levers are in the same state.
-

Conclusion: Your Journey Into Optimization

You now know the difference between a circuit designed for teaching and one designed for practical builds. This is a major step in your engineering journey. You understand that compact designs are not magic; they are just clever physical implementations of the same Boolean logic you have already mastered.

From Module 3 onward, we follow the **Rule of Abstraction**:

- A gate is defined by its **truth table** and interface, not by its internal layout.
- You are free to use verbose or compact designs as you prefer.

Explore More: The Gate Museum

In the world download provided for the course, you will find a section labeled "Gate Museum" which showcases these and many other community-tested compact designs for

each logic gate. I encourage you to explore, build, and test them to expand your engineering toolkit.

Module 3: From Binary to Pictures - Building a Digital 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. This is our first major engineering project, where we'll turn abstract binary signals into a number we can actually read.
 - **Learning Goals:**
 - Understand the distinct roles of a decoder and an encoder.
 - Grasp the engineering trade-offs between a "brute-force" design and an elegant, compact design.
 - Master "active-low" logic and its practical application in Redstone.
 - Build a functional Diode Matrix and understand its role as a form of Read-Only Memory (ROM).
 - **Lesson Overview:**
 - **Lesson 3.1:** The Goal: Building Our 7-Segment Display
 - **Lesson 3.2:** The Master Plan: A Two-Stage Translation
 - **Lesson 3.3:** The Decoder Lab: A Simple "Brute-Force" Build
 - **Lesson 3.4:** The Decoder Solution: An Elegant, Compact Design
 - **Lesson 3.5:** The Encoder: Building a "Diode Matrix" ROM
 - **Lesson 3.6:** The Grand Payoff: The Final Connection
 - **Lesson 3.7:** 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 about building.

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.

Lesson 3.1: The Goal: Building Our 7-Segment Display

Key Takeaway: A 7-segment display is a standard output device that uses seven independent segments to form numbers. Understanding how to control it manually is the first step to controlling it automatically.

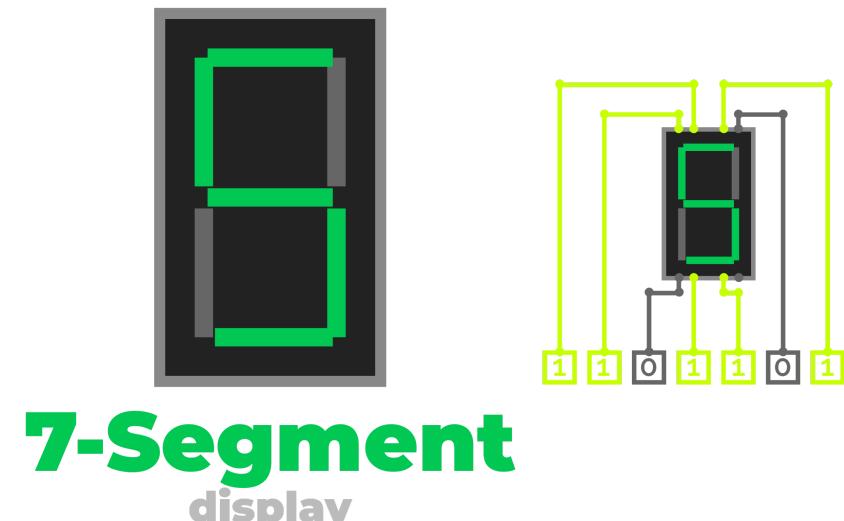


Figure: The symbol for a 7-Segment Display on CircuitVerse (left) and its function in a basic circuit (right), taking seven inputs and lighting up the segments based.

Our computer can hear us, but it can't talk back. So far, all our work is invisible, buried in wires and circuits. How do we make our computer show us numbers in a way we understand?

The answer is the **7-segment display**, a classic output device found in everything from digital clocks to microwaves. It uses seven independently controlled segments, labeled `a` through `g`, arranged in an '8' pattern.

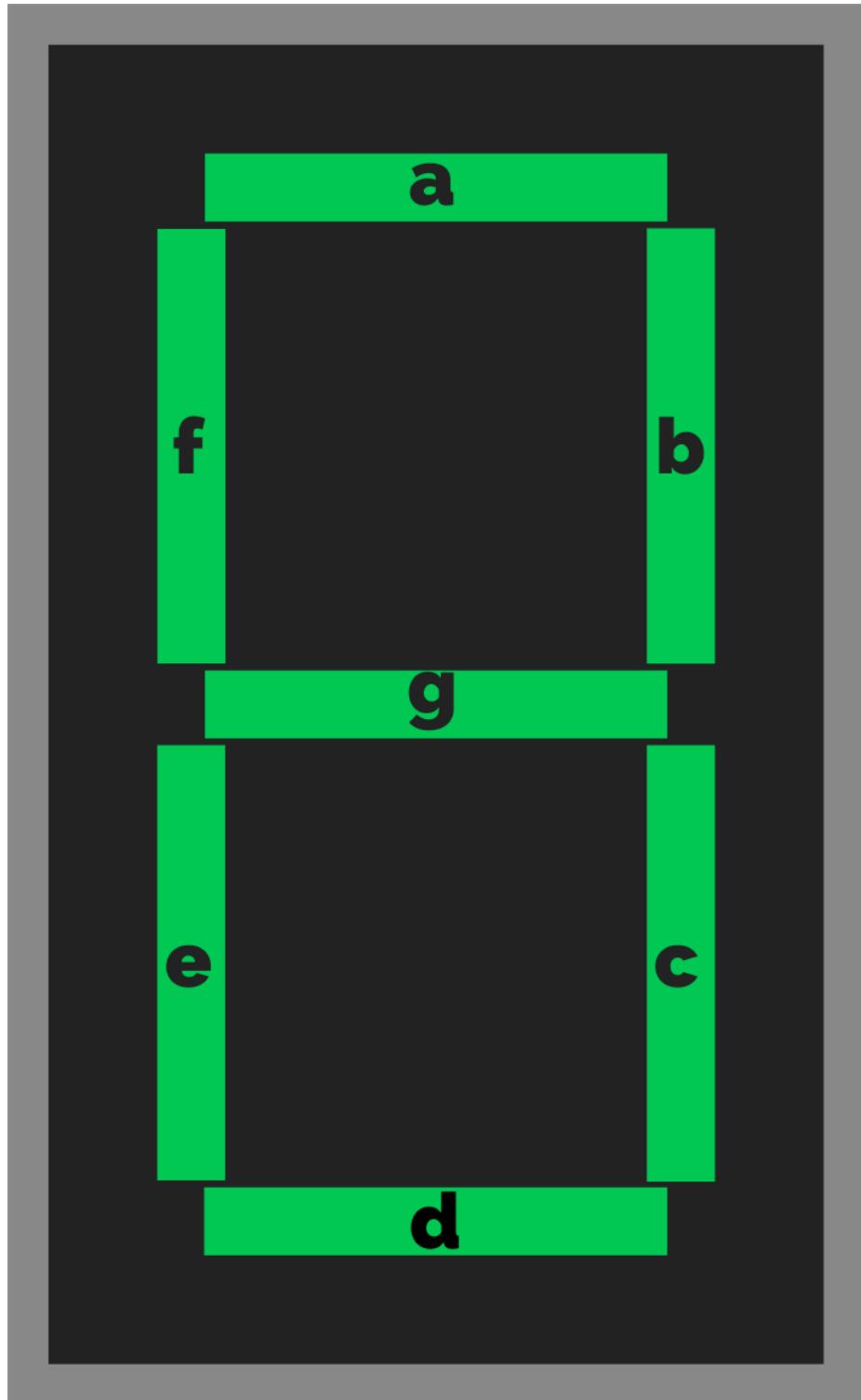


Figure: The standard labeling for the segments of a 7-Segment Display..

By lighting up specific combinations of these seven segments, we can display any digit from 0 to 9.

Lab: Building the Physical Display

Let's start by building the physical canvas for our numbers.

- 1. Construct the Segments:** In Minecraft, place Redstone Lamps in the "8" shape shown above. For good visibility, making each segment 3 lamps long is a great choice.

2. Isolate the Segments: Carefully surround the lamp segments with a non-conductive block like Wool or Concrete. I use black concrete to make the segments stand out.

3. Create Manual Controls: To power each segment, run a Redstone Repeater into the middle lamp. For now, place a solid block behind each repeater and attach a Lever to it. This gives you manual control for testing.

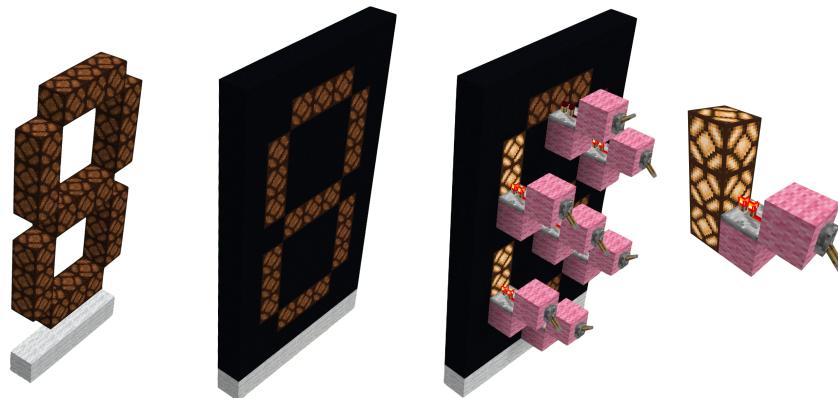


Figure: The display's construction stages. From left to right: the basic lamp layout, the layout isolated with concrete, powering the middle lamps of each segment, and a close-up of the repeater and lever used to control a single segment.

Practice Lab: Becoming a Human Encoder

Before we build the complex logic to control this display automatically, let's get a feel for it ourselves. Use the levers you just installed to "draw" the following digits. This exercise will build your intuition for exactly what our machine needs to accomplish.

Note: The levers are on the back of the display, so keep that in mind when flipping specific segments. It might help to label the segments with a sign by the lever that controls it for this exercise.

1. Flip the levers for segments **b** and **c**. You should see the digit **1**.
2. Now, try to display the digit **7**. (You'll need segments **a**, **b**, and **c**).
3. Next, create the digit **4**. (This requires segments **f**, **g**, **b**, and **c**).
4. **Challenge:** Try to form the digit **8**. What do you notice? Now try to form the digit **2**.

Lesson 3.2: The Master Plan: A Two-Stage Translation

Key Takeaway: Complex engineering problems are best solved by breaking them down into smaller, simpler, manageable stages. The "plan" for our encoder is essentially a lookup table.

Now that we have our display, how do we control it? Our computer thinks in 4-bit binary, but our display needs 7 separate signals. Connecting the 4-bit input directly to the 7 segments would be a nightmare.

Instead, let's think like engineers and break the problem into two much simpler, more manageable stages:

1. **Decoder:** This first stage will act as an "identifier". Its only job is to look at the 4-bit binary input and determine *which* number (0-9) it represents. It will then activate a single, unique output line corresponding to that number.
2. **Encoder:** This second stage is will act as the "mapper". It receives the simple signal from the decoder (e.g., "the number is 3!") and "maps" the signal, and the number it represents, to the correct combination of the 7 segments.

This modular, two-stage approach is the heart of good engineering. It's easier to build, easier to test, and far easier to fix if something goes wrong.

Our Signal Flow: [4-bit Input] → [**Decoder**] → [1 of 10 Lines] → [**Encoder/ROM**] → [7 Segment Signals] → [Display]

Lesson 3.3: The Decoder Lab: A Simple "Brute-Force" Build

Key Takeaway: A decoder can be built by assigning one AND gate to recognize each unique binary input. This "brute-force" method is clear but does not scale well.

Before we tackle our full 4-bit to 10-line decoder, let's build a smaller, simpler version to prove the concept. We are going to build a **2-bit to 4-line decoder**. This circuit will take a 2-bit binary input (00, 01, 10, 11) and light up one of four corresponding output lamps (L1, L2, L3) representing those values in decimal (0, 1, 2, 3).

By scaling down the problem, we can focus on the core logic without getting overwhelmed. This is a common engineering practice: start small, prove the concept, then scale up. I'm calling this a "brute-force" method because we will build a separate AND gate for each output, rather than using a more elegant design, which we will learn in the next lesson.

The Logic on Paper

- **Inputs:** B1 (the "2s" place), B0 (the "1s" place)
- **Outputs:** L0, L1, L2, L3
- **Logic Gates:** We need one 2-input AND gate for each output.
 - L0 (for 00 or 0) = (!B1) AND (!B0)
 - L1 (for 01 or 1) = (!B1) AND B0
 - L2 (for 10 or 2) = B1 AND (!B0)
 - L3 (for 11 or 3) = B1 AND B0

Lab: Building the 2-to-4 Decoder

Step 1: The 2-Bit Bus

1. Set up two the standard inputs we've been using throughout the course, the redstone lamp with a lever on one side. Label them `B1` and `B0`.
2. From these levers, create a **4-line bus**. Run redstone dust from the back of each lamp to a central point and then split each line into two parallel lines. On one line of each pair, place a NOT gate (a block with a torch on the opposite side that the dust runs in to).
3. You now have four parallel lines carrying the signals `B1`, `!B1`, `B0`, and `!B0`. Use colored wool to keep them organized!

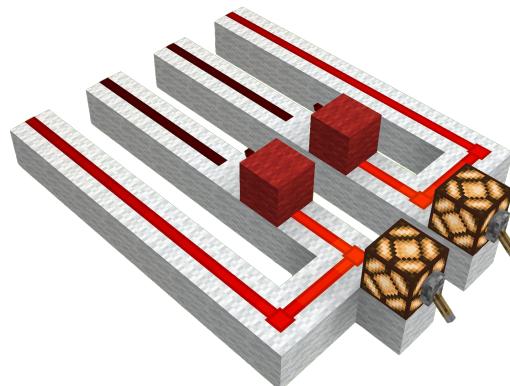


Figure: 4-line bus with inputs 'B1' and 'B0' and their inversions

Step 2: Build and Test the First Gate (`L0`)

1. Choose your favorite 2-input AND gate design from Module 2 or the Interlude. Build one of these gates.
2. Connect the gate's two inputs to the `!B1` line and the `!B0` line on your bus. Be careful with your wiring!
3. Place a Redstone Lamp at the output of the AND gate. This is your `L0` output.
4. **Test it!** Set your input levers to `00` (`B1` =OFF, `B0` =OFF). The `L0` lamp should turn ON. Now, flip either lever. The lamp should turn OFF. This proves your first gate is wired correctly.

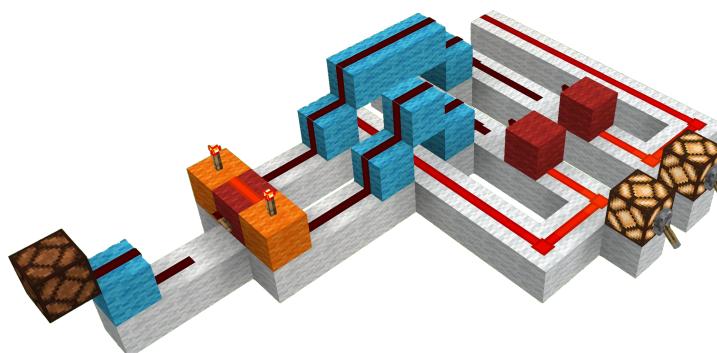


Figure: Single AND gate connected to the '`!B1`' and '`!B0`' lines of the bus. The input is set to '`11`', so the '`L0`' lamp is OFF. It would be on if the input were '`00`'.

Step 3: Build the Remaining Gates

1. Build three more identical 2-input AND gates next to the first one.
2. Wire them according to the logic table:
 - **Gate for L1** : Connect its inputs to the `!B1` and `B0` bus lines.
 - **Gate for L2** : Connect its inputs to the `B1` and `!B0` bus lines.
 - **Gate for L3** : Connect its inputs to the `B1` and `B0` bus lines.
3. Place a Redstone Lamp on the output of each gate.

Step 4: The Grand Test

Now, cycle through all four possible inputs with your levers:

- `00` -> Only the `L0` lamp should be ON.
- `01` -> Only the `L1` lamp should be ON.
- `10` -> Only the `L2` lamp should be ON.
- `11` -> Only the `L3` lamp should be ON.

Congratulations, you've built a working decoder!

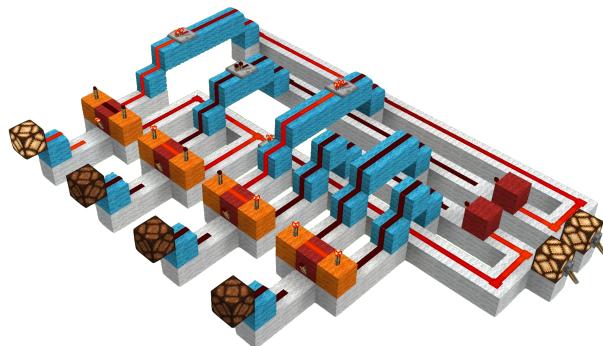


Figure: Final working 2-to-4 decoder, with the input set to '11', so only the 'L3' lamp is ON.

Lesson Summary: The Problem of Scale Take a look at the space your 2-to-4 decoder occupies. Now, imagine our real goal: a 4-to-10 decoder. We would need **ten** 4-input AND gates, which are much larger than the simple gates we just used. The brute-force method works, but it does not scale well. It creates a massive, resource-hungry machine.

In the next lesson, we will learn a far more elegant and compact solution.

Lesson 3.4: The Decoder Lab, Part 2: An Elegant, Compact Solution

Key Takeaway: By using an "active-low" design and two clever types of "taps" (Repeater and Torch), we can build a decoder that is vastly smaller and more efficient.

Welcome to the engineer's solution. Instead of an "active-high" design, we will build an "**active-low**" design where the correct line turns **OFF**.

The Core Concept: The Mismatch Detector

Each output line will function as a "**mismatch detector**." Its job is to power its own wire (turning its lamp OFF) if the input does **not** match the line's identity. The only time a lamp stays ON is when the input is a perfect match. A "tap" is simply our term for a connection that reads, or "taps into," the signal from one of the main bus lines.

Technically, the entire structure for each output line is a **Multi-Input NOR Gate**, but thinking of it as a "mismatch detector" is a great way to understand its function.

Two Types of Taps: The Key to the Design

We use two different methods to tap the bus. This clever approach allows a single bus line (e.g., `B1`) to do the work of the two separate `B1` and `!B1` lines we needed in the brute-force build, cutting our bus width in half.

1. **The Repeater Tap (Checks for a `1`):** A Repeater placed to tap a bus line will only activate if that bus line is **ON** (`1`). We use this to detect a `1` where we expect a `0`.
2. **The Torch Tap (Checks for a `0`):** A Torch placed to tap a bus line will only activate if that bus line is **OFF** (`0`). We use this to detect a `0` where we expect a `1`.

The Simple Rule for Building

To program the wire for an output line `LN` :

- For every bit position that is `0` in its identity, place a **Repeater Tap**.
- For every bit position that is `1` in its identity, place a **Torch Tap**.

Lab & Experiment: Building the Compact 4-to-10 Decoder

The Setup: Building the Physical Structure

This design relies on a two-layer structure to keep the input and output lines separate.

1. **Output Layer (Ground Level):** Lay out 10 parallel lines of Redstone dust for your output lines (`L0` through `L9`). Leave at least one empty block between each line to prevent interference. At the end of each line, place a solid block, a Redstone torch on top, and a Redstone Lamp on top of the torch. All 10 lamps should be ON by default.

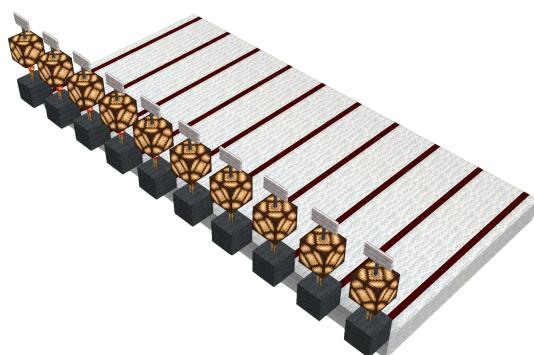


Figure: Screenshot showing the 10 output lines on the ground, step 1 of the compact 4-to-10 decoder

2. **Input Layer (Floating):** Now, build a platform for your input bus two blocks off the ground (leaving a 1-block high air gap). On this platform, run your four parallel input bus lines (`B3` to `B0`) so they run perpendicularly across all 10 output lines below.

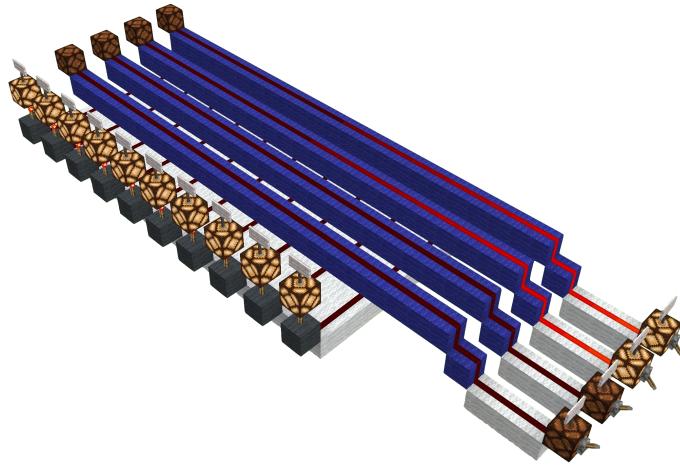


Figure: The two-tiered structure with four input bus lines ('B3' to 'B0') floating above the 10 output lines.

Programming the Lines: Placing the Taps

Now we will place our taps to connect the input and output layers, “programming” each output line to detect its unique binary identity. This is where the active-low logic comes to life. Each tap checks for a mismatch, and only the perfectly matched line stays unpowered (lamp ON).

- **How to Build a Torch Tap:** At the correct intersection, place a Redstone torch on the side of the block that the input bus line rests on, directly above the output wire below. This tap activates (powers the output wire) when the bus line is OFF (`0`).
- **How to Build a Repeater Tap:** This requires specific placement to achieve strong power. At the correct intersection, one block *before* the output wire, break the input bus line. On the ground level, place a solid block and put a Repeater on top of it, facing in the direction of signal flow. This "snaking" path is essential. It is important to note that the Repeater itself does not power the output wire directly; it powers the block it runs into, which then becomes strongly powered and can power the output wire.

Let’s apply this to one line to see it in action, then you’ll program the rest using the reference chart.

Programming Example: Line L3 (Identity: `0011`)

To make the `L3` line detect the binary input `0011` (decimal 3), we need to place taps according to its identity:

- `B3` is `0` : Place a **Repeater Tap** (checks for a `1`, powers the wire if mismatched).
- `B2` is `0` : Place a **Repeater Tap**.
- `B1` is `1` : Place a **Torch Tap** (checks for a `0`, powers the wire if mismatched).

- `B0` is `1` : Place a **Torch Tap**.

Here's what it looks like once you've added the taps for `L3` :

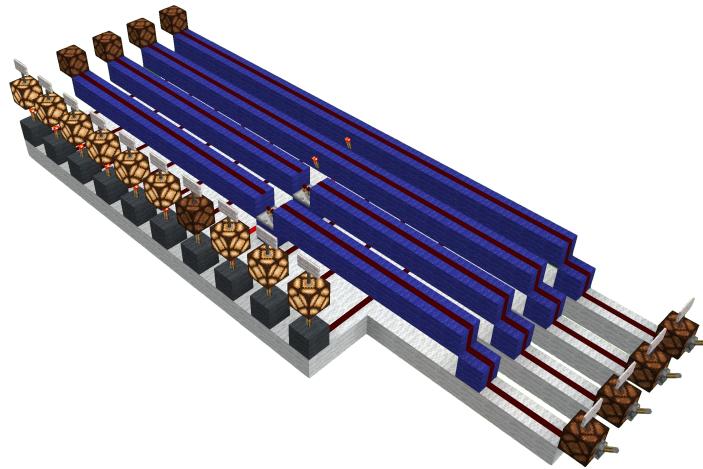


Figure: The two-tiered structure with taps added for the 'L3' line, set to input '0000'. All lamps are lit except 'L3', which is off due to its mismatch detectors activating.

With the taps in place, test the `L3` line by setting the input to `0000` (all levers OFF). Every output line except `L3` should have at least one tap activated (powering the wire, turning the lamp OFF). For `L3`, all taps are inactive because the input doesn't match `0011`, so its lamp stays ON. This confirms your mismatch detector is working!

To get a closer look at how the taps are placed, check out this isolated view of the `L3` line:

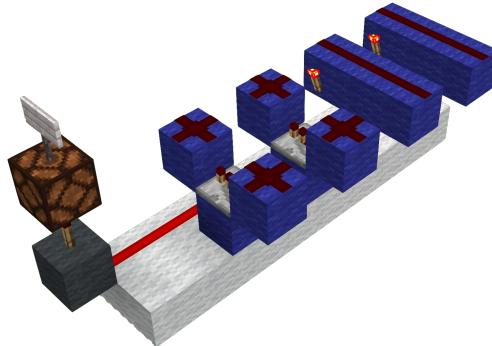


Figure: Close-up of the 'L3' line with two Repeater Taps ('B3', 'B2') and two Torch Taps ('B1', 'B0'), no inputs active.

This zoomed-in view shows exactly where to place each tap for `L3`. Notice the “snaking” path of the Repeater Taps, ensuring strong power, and the Torch Taps hanging off the side of the input bus blocks. Precision here is key! Double-check your placements to avoid crossed signals.

To verify the `L3` line works as intended, you can add levers to test it independently before connecting all lines. Set the inputs to `0011` (matching `L3`'s identity):

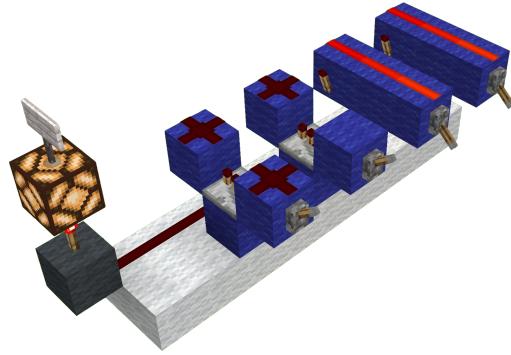


Figure: Isolated 'L3' line with levers set to '0011', lighting the 'L3' lamp to confirm correct tap placement.

In this test, the levers mimic the input `0011`. The `L3` lamp lights up because no taps activate (no mismatches), leaving the wire unpowered. Try flipping any lever (for example, `0010`), and the lamp should turn OFF as a tap detects a mismatch. This hands-on test builds confidence before scaling to all 10 lines.

Complete All Lines: Using the Reference Chart

Apply the rule and build methods to the remaining 9 lines. Use the chart below to verify your placements. This is your blueprint.

Bus Line	L0	L1	L2	L3	L4	L5	L6	L7	L8	L9
B3 (8)	R	R	R	R	R	R	R	R	T	T
B2 (4)	R	R	R	R	T	T	T	T	R	R
B1 (2)	R	R	T	T	R	R	T	T	R	R
B0 (1)	R	T	R	T	R	T	R	T	R	T
<i>(R = Repeater Tap, T = Torch Tap)</i>										

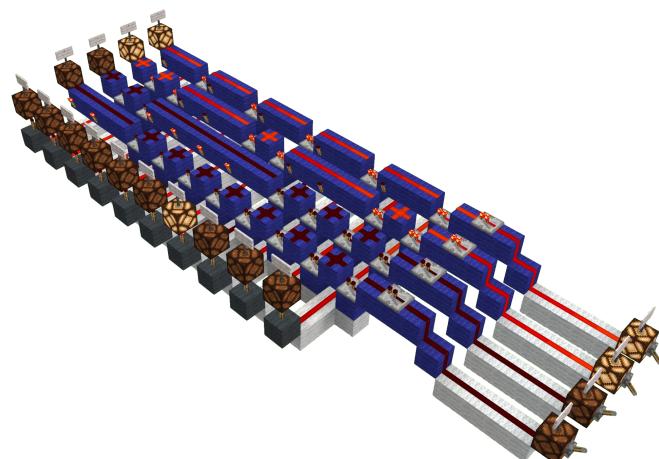


Figure: The complete 4-to-10 compact decoder in action, with input 0011 lighting only the L3

Test Your Work!

Cycle through inputs `0000` to `1001`. Verify that only one lamp is lit for each input.

Practice Problems

Problem 1: Design on Paper

Before you build, an engineer must be able to plan. For output line `L6` (**Identity:** `0110`), what taps would you need? List out which type of tap (Repeater or Torch) is required for each of the four bus lines (`B3`, `B2`, `B1`, `B0`).

(Solution for this exercise is in Appendix B)

Problem 2: The Debug Challenge

You've built your decoder, but something is wrong. When you set the input levers to `1001` (for the number 9), you notice that the lamp for `L9` is on (which is correct), but the lamp for `L8` is also on (which is incorrect).

What is the single most likely mistake in your build that would cause this specific error?

(Solution for this exercise is in Appendix B)

Lesson 3.5: The Encoder: Programming a "Diode Matrix" ROM

Key Takeaway An encoder can be built as a physical Read-Only Memory (ROM) using a "diode matrix," where the layout of the wiring permanently stores the data for how to draw each number.

We now have a working decoder that gives us a single **unpowered** (active-low) line for any given number. The next step is to build our "mapper", the encoder that will take this single signal and draw the correct digit on our display.

The Concept: A Physical Lookup Table

This stage is effectively a physical **Read-Only Memory (ROM)**. The "address" is the active-low line from the decoder, and the "data" that it looks up is the pattern of segments for that number. We will build this using a structure called a **Diode Matrix**.

First, let's create the plan on paper. This lookup table is the blueprint for our build.

7-Segment Display Segment Table

Digit	a	b	c	d	e	f	g

Digit	a	b	c	d	e	f	g
9	X	X	X	X		X	X
8	X	X	X	X	X	X	X
7	X	X	X				
6	X		X	X	X	X	X
5	X		X	X		X	X
4		X	X			X	X
3	X	X	X	X			X
2	X	X		X	X		X
1		X	X				
0	X	X	X	X	X	X	
<i>(X = segment ON)</i>							

The Logic: Inverting the Inversion

This is where our active-low signal becomes very powerful.

- Our input is a single LOW line from the decoder.
- Our goal is to turn this one LOW signal into multiple HIGH signals to power the correct display segments.
- We can do this perfectly with **Redstone Torches**. When the input line from the decoder is LOW, any torch placed along it will turn **ON**.

This gives us a very simple rule: to turn a segment ON for a given number, we just need to place a torch at the intersection of that number's line and that segment's line.

Lab & Experiment: Building the Diode Matrix

1. The Setup: The Output Layer

Start by building the foundation for your Diode Matrix: the output lines that will control the 7-segment display.

- **Segment Output Layer (Ground Level):** Lay out 7 parallel lines of Redstone dust, one for each segment (`a` through `g`). These will carry signals to the display. Leave a 1-block gap between each line to prevent interference. Add Redstone Repeaters every 15 blocks to keep the signals strong, as these lines may need to travel to your display.



Figure: The 7 parallel segment output lines ('a' through 'g') on the ground, with repeaters for signal strength.

This ground layer is the backbone of your encoder, carrying the signals that will light up the display segments. Double-check that each line is isolated to avoid crossed signals.

2. The Grid: Adding the Input Layer

Now, add the input layer to complete the Diode Matrix grid. Eventually these lines will connect to the decoder's active-low lines.

- **Decoder Input Layer (Floating):** Build a platform of solid blocks one level directly above the ground layer (no air gap). On this platform, run 10 horizontal lines of Redstone dust for the decoder outputs (`L9` down to `L0`), perpendicular to the 7 segment lines below. Place a Redstone Lamp at the end of each input line to visualize which line is active (LOW).



Figure: The two-layer Diode Matrix structure, with 7 segment output lines on the ground and 10 input lines ('L9'–'L0') above, lamps showing input activity.

This two-layer grid is your ROM's framework. The lamps are optional, but give a nice visual for what is happening. When a lamp is ON, its line is LOW (active). Take a moment to admire the clean, perpendicular layout as it is the key to programming the segment patterns efficiently.

3. Programming the Matrix: Placing the Torch Taps

Now, you'll "burn" the lookup table into the hardware by placing torch taps at the correct intersections. This is where you physically encode the segment patterns for each digit.

- **The Rule:** For each number line `LN`, consult the lookup table. For every segment that should be **ON** for that number, place a torch tap.
- **How to Build the Tap:** At the correct intersection, place a **Redstone Torch on the side of the block** that the horizontal input line (`LN`) rests on. Position the torch to power the segment line on the ground below.

Programming Example: Line `L9`

Let's program the `L9` line (digit 9) as an example. According to the lookup table, digit `9` needs segments `a, b, c, d, f, g` to be ON. Place six torch taps along the `L9` line, one at each intersection with those segment lines.

Here's a close-up of the `L9` line with its taps in place:

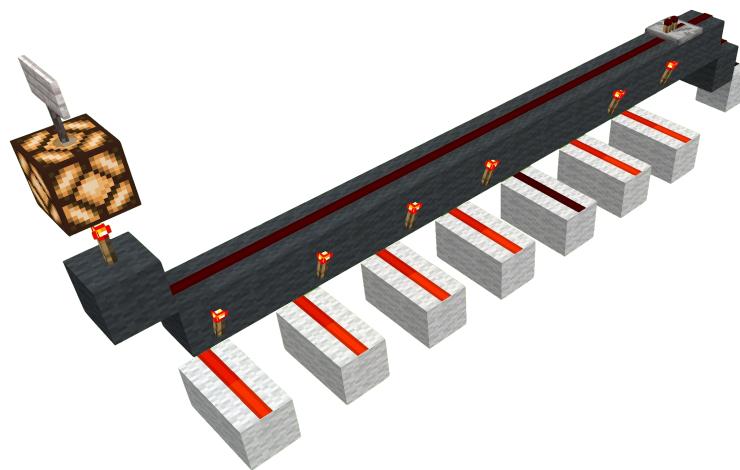


Figure: Close-up of the '`L9`' line with six torch taps programming segments '`a, b, c, d, f, g`' for digit 9.

This zoomed-in view shows exactly where to place the torch taps for `L9`. Each torch is attached to the side of the block supporting the `L9` line, powering the segment lines below (`a, b, c, d, f, g`). These torches are your ROM's "data" by each representing a specific segment that lights up when `L9` goes LOW. To test it, place a lever at the start of the `L9` line and set all other lines to ON (using levers). When you turn the `L9` lever OFF (simulating the decoder's active-low signal), the `L9` lamp should light up, and the segment lines `a, b, c, d, f, g` should activate. You can place temporary redstone lamps at the segment line ends to verify. If any segment doesn't light, double-check your torch placements against the lookup table.

Complete the Matrix

Repeat this process for all 10 lines (`L0 – L9`), using the lookup table to place torches for each digit's segment pattern. Work methodically to avoid mistakes. This is like programming a game cartridge, where every torch is a bit of stored data. Precision is key to ensure each line activates the correct segments.

Engineering Note: The Self-Isolating Design

You might wonder if we need repeaters to isolate the segment lines from each other like we did in our basic OR gate. In this specific design, we don't! The Redstone Torches we use as taps are naturally **one-way devices**. They send power *out* to the segment line, but power from another torch cannot flow *backwards* through them. The torches act as the diodes in our "Diode Matrix," making the design incredibly elegant and efficient.

4. Test Your Work

Before connecting the encoder to the decoder, test all lines (`L0 – L9`) independently, as you did for `L9`. Place a lever at the start of each line, set all others to ON, and turn the tested line OFF. Verify that the segment patterns match the lookup table (e.g., `L3` should light `a, b, c, d, g` for digit 3). Here's what the fully programmed Diode Matrix looks like:

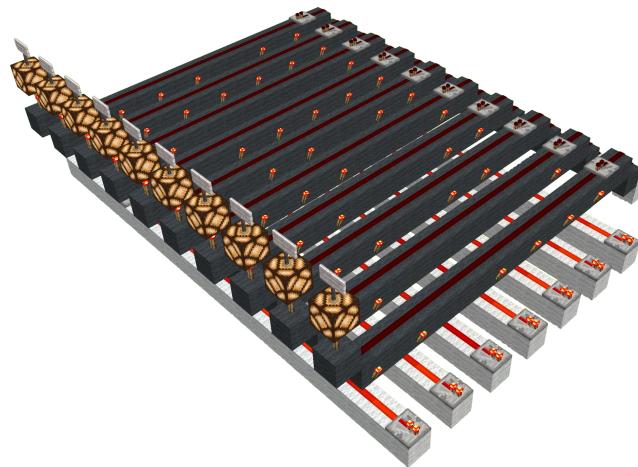


Figure: The complete 10-to-7 encoder with all torch taps placed, showing the 'L3' line active (input '0011') and segments 'a, b, c, d, g' powered for digit 3.

This is your finished ROM, with every line programmed to map decoder inputs to segment outputs. The `L3` line is active here (LOW), lighting up the correct segments for a "3." Cycle through inputs `L0 – L9` to confirm each digit's pattern. If any segments don't light as expected, revisit your torch placements using the lookup table. You've just built a physical memory that "stores" the display patterns for all 10 digits!

Real-World Connection: BIOS and Game Cartridges

The "Diode Matrix" you've just built is a simple but powerful form of **Read-Only Memory (ROM)**. The "program" is physically burned into the circuit's layout by the placement of the torches. This exact principle was fundamental to early computing. A computer's **BIOS chip**, which tells it how to boot up, is a form of ROM. Old video game cartridges were also ROMs, with the entire game's data permanently stored in the hardware's structure. You've built the same technology!

Practice Problems

Problem 1: Design on Paper

You are programming the line for the digit **2**. According to the lookup table, which perpendicular segment lines need a torch tap from the horizontal **L2** line?

(Solution for this exercise is in Appendix B)

Problem 2: The Debug Challenge

When you test your encoder by providing a LOW signal to the **L4** line, you expect to see the digit **4** (segments **b, c, f, g**). Instead, the display shows **b, c, f** but **segment g remains dark**. What is the most likely cause of this error?

(Solution for this exercise is in Appendix B)

•

Lesson 3.6: The Grand Payoff: System Integration

Key Takeaway Connecting individual, tested modules into a complete, working system is the final and most rewarding step of any engineering project.

The moment of truth has arrived. You've built and tested the decoder to identify numbers, the encoder to map them to segment patterns, and the 7-segment display to show the results. Now, it's time to connect these modules and watch your digital display come to life, transforming binary inputs into human-readable digits. Taking modular pieces and creating a cohesive system, this is engineering at its finest!

Lab & Experiment: The Final Connection

This final step is all about making the connections between all of the components from this module. The wiring may get a bit messy, but as long as the signals flow correctly, you are good to go!

- 1. Connect Decoder to Encoder:** Carefully connect the 10 active-low output lines from your **Decoder** (**L0 – L9**) to the 10 horizontal input lines of your **Encoder/ROM**. Use Redstone Repeaters as needed to ensure the signals remain strong over long distances. Label your lines (e.g., with colored wool) to avoid mix-ups.
- 2. Connect Encoder to Display:** Connect the 7 output lines from your **Encoder/ROM** (**a – g**) to the control inputs of the **7-Segment Display** you built in Lesson 3.1. This may require creative wiring to route signals to the display's repeaters, but ensure each segment line connects to its corresponding input (e.g., **a** to the **a** segment). Test each connection with a temporary lever to confirm the segment lights up.

Here's what your fully connected system should look like, with the input set to **0011** to display a "3":

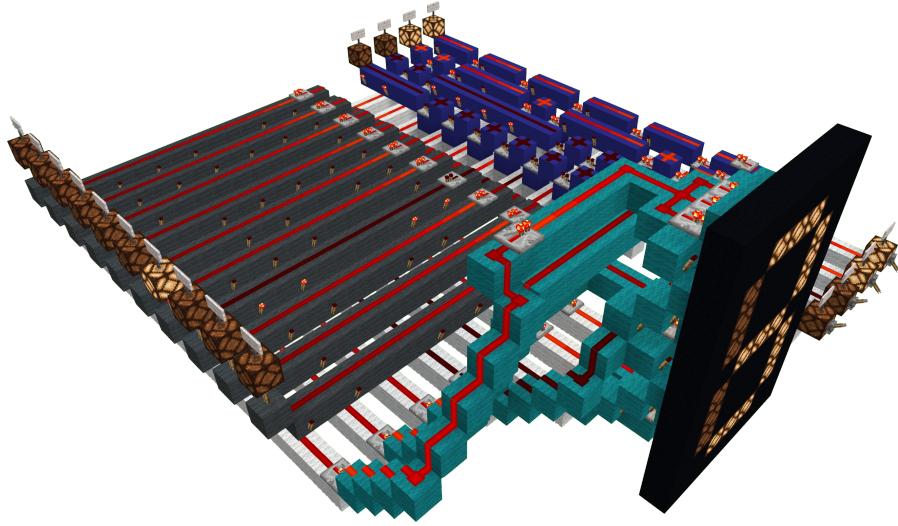


Figure: The complete digital display system in action, with input `0011` activating the `L3` line and lighting segments `a, b, c, d, g` to form a glowing “3”.

Take a moment to admire this masterpiece! Your modular design has paid off, making this complex system manageable and functional.

Let's Trace the Signal: 3 (0011)

To solidify your understanding, let's trace the signal through the entire system with the input set to **0011** (decimal 3):

1. You flip the input levers to **0011** ($B_3=0, B_2=0, B_1=1, B_0=1$).
2. **In the Decoder:** The mismatch detector for the **L3** line (identity **0011**) finds a perfect match. All its taps (Repeaters on **B3**, **B2**; Torches on **B1**, **B0**) are OFF, so the **L3** wire becomes **unpowered (LOW)**. Every other line (**L0 – L2**, **L4 – L9**) has at least one tap activated, powering their wires **HIGH**.
3. **In the Encoder:** The HIGH lines keep their torches off. The **L3** line, being LOW, turns ON the torches at its intersections with segments **a, b, c, d, g** (per the Lesson 3.5 lookup table).
4. Those five torches send power down their respective segment lines.
5. **At the Display:** The signals reach the 7-segment display, lighting up segments **a, b, c, d, g** to form a perfect “3”.

From above, you can see how compactly your system fits together:

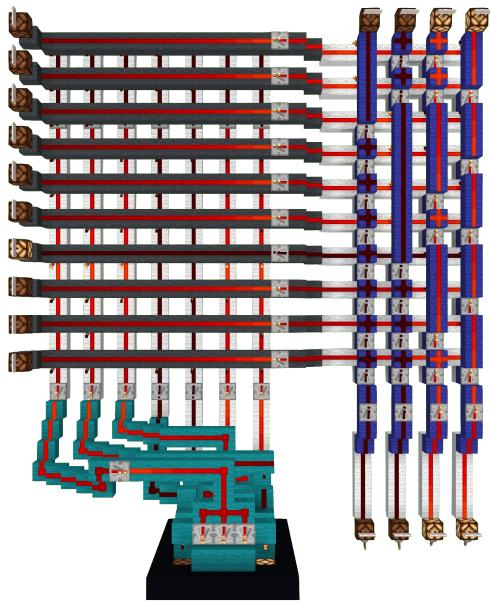


Figure: Aerial view of the compact digital display system, with input `0011` producing a “3”. The modular layout connects the decoder, encoder, and display efficiently.

This top-down view highlights the elegance of your modular design. The decoder’s input bus, the encoder’s torch matrix, and the display’s segments are tightly packed yet clearly organized. While the torches in the encoder grid are less visible from this angle, refer to the Lesson 3.5 lookup table to confirm their placements. Cycle through inputs `0000` to `1001` and watch the display light up each digit perfectly.

Congratulations! You’ve engineered a complete system that translates 4-bit binary into human-readable digits. This is a massive milestone in digital electronics, and you should be proud of your work. Your ability to break down a complex problem into modular components and wire them together is a hallmark of great engineering.

Lesson 3.7: Module 3 Checkpoint

This checkpoint is divided into three parts to test the different skills you've acquired.

Part 1: Knowledge Check

1. Why is a two-stage (Decoder -> Encoder) design generally better than a single, complex circuit?
2. What is the purpose of the **Repeater Tap** in our compact decoder? Why can't we just use Redstone dust?
3. In our Diode Matrix ROM, what does placing a **Torch Tap** at an intersection physically represent?

(Solution for this exercise is in Appendix B)

Part 2: Logic Puzzles

- Decoder Design:** You want to add a special output line, `LE`, that lights up only for even numbers (`0, 2, 4, 6, 8`). You realize that for all even numbers, the `B0` bit is always `0`. What is the single tap you would need to build a simple detector for this?
- Encoder Design:** The letter 'A' can be made with segments `a, b, c, e, f, g`. According to the design of our ROM, which segment line is the *only one* that would **not** have a torch tap placed on it from the `LA` input line?
- Reverse Engineering:** You see a line in a decoder that has Torch Taps on `B2` and `B1`, and Repeater Taps on `B3` and `B0`. What decimal number is this line designed to detect?

(Solution for this exercise is in Appendix B)

Part 3: The Debug Challenge (In-Game)

In the world download for this module, you will find a section labeled "Module 3 Debug Challenge." The display system is fully connected. When you input `0010` (for the number 2), the display incorrectly shows a `6`.

Trace the logic:

- The digit `2` should be `a, b, g, e, d`.
- The digit `6` is `a, c, d, e, f, g`.

What is the single most likely point of failure in the system that would cause this specific error?
(Hint: The problem is in the Encoder/ROM).

(Solution for this exercise is in Appendix B)

Module 3 Conclusion

This was a massive milestone. You didn't just build a circuit; you engineered a complete system. By breaking a complex problem down into distinct, logical stages, you built something complex in a way that was manageable, testable, and understandable. You have now mastered the concepts of binary-to-decimal decoding and using a hardware ROM to drive an output, two fundamental building blocks of digital electronics.

What's Next? You have successfully completed Part I of this course. You can now take a binary input and display it as a number humans can read. But what happens when we try to do math? In the next module, you'll discover a critical flaw in our simple translator when we try to count past 9. You'll learn about the hexadecimal system and how our modular design makes upgrading our system a breeze.

Key Terms (Module 3)

- 7-Segment Display:** An arrangement of seven light segments that can be combined to display numbers and some letters.

- **Active-Low Logic:** A design principle where the "active" or "on" state is represented by a LOW (unpowered) signal.
- **BCD (Binary-Coded Decimal):** A method of representing the decimal digits 0-9 using a 4-bit binary code.
- **Decoder:** A circuit that takes a multi-bit binary input and activates a single, corresponding output line. Our decoder acts as an **Identifier**.
- **Diode Matrix:** A grid of input and output lines where components (like our taps) are placed at intersections to create a programmable logic device, often used as a ROM.
- **Encoder:** A circuit that takes a single active input line and translates it into a multi-bit coded output. Our encoder acts as a **Mapper**.
- **Modularity:** The engineering practice of designing a system in independent, interchangeable components. This makes the system easier to design, test, and upgrade.
- **ROM (Read-Only Memory):** A type of storage where data is permanently programmed into the hardware's structure.
- **Tap (Repeater/Torch):** Our term for a connection that reads a signal from a bus line to control another wire.

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.