The CPU

A very tall pile of simple

Ekansh Goenka

bytecolony.computer

Copyright © 2026 Ekansh Goenka. All rights reserved.
First edition, 2026.

The web edition of this book, including animated versions of the diagrams, is available at bytecolony.computer.

Cover design by Ekansh Goenka.
Typeset in TeX Gyre Pagella, with IBM Plex Mono for code.

Contents

Part 1: Foundations **1**

1. How to Read This 3
2. Circuits & Switches 3
3. Relays & Logic Gates 8
4. Counting With Wires 18
5. Making 1 + 1 = 10 22

Part 2: Machinery **35**

6. Storing a Bit 35
7. Taking the Bus 44
8. Organizing Data 51
9. The ALU 61
10. The Big Picture 65

Part 3: Execution **71**

11. Instructions Are Numbers 71
12. The Not-So-Clever Control Unit 77
13. Decode & Execute 82
14. Finally, the Fun Part 91
15. The End 96

Appendix: Quick Reference **98**

Glossary **100**

Further Reading **103**

Acknowledgments **104**

About the Author



Ekansh Goenka (b. 2013) grew up in Singapore. He got interested in computers at eight, thanks to his dad's techy influence. He started out making apps and websites, and curiosity slowly led him down the stack. He is also interested in robotics, embedded systems, and computer science in general. Outside of computers, he likes reading, playing piano and guitar, and Brazilian jiu-jitsu.

He wrote this book in 2026, over about six months. He drew all 71 diagrams by hand and designed its 8-bit CPU from scratch. He wrote it because once he understood how a CPU works in detail, it turned out to be simpler than most people think. The resources he found either skipped the juicy details, so you never really got it, or were so long and intimidating that most people never started. The web version of this book is free at bytecolony.computer, his personal website, with the animated diagrams.

Part 1: Foundations

You can hear the phrase "computers think in 1s and 0s" a hundred times and still not understand how a computer actually works. By itself, this explains basically nothing. Sure, a wire can be high or low. Sure, a light can be on or off. But how does that become addition?

How does that become memory?

How does that become a program sitting in RAM, one instruction after another, telling a machine what to do?

Some resources stay extremely high-level, so you never really understand how a CPU actually works. The deeper resources are amazing, but they are long, dense, and intimidating. And frankly, for someone who doesn't want that level of detail, a lot of it can often feel like too much.

My goal is to help you understand what is going on under the hood, without exploding your brain or eating weeks of time.

We start with a simple circuit turning a light bulb on and off, then work our way through logic gates, memory, and the basic circuits underneath them.

The key point is that nothing here is smart in isolation. A CPU is not one hard idea. It is a very tall pile of simple ones.

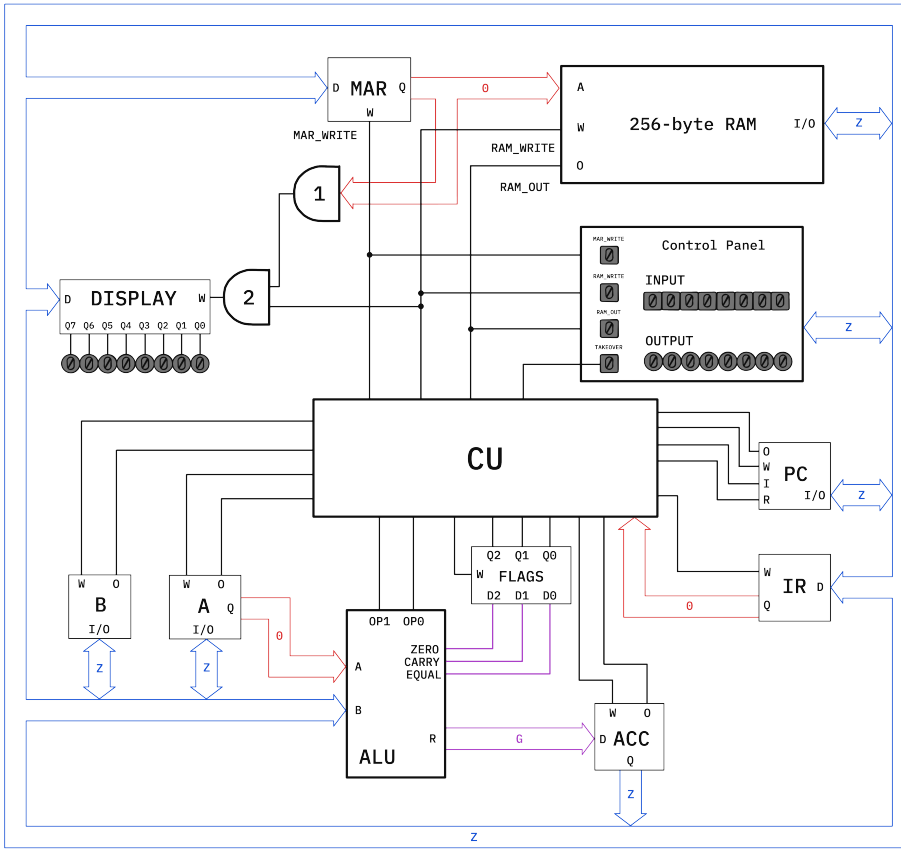


Diagram 0.1. The whole computer.

This probably looks like an overwhelming mess, but don't worry, that's normal. Throughout this book we are going to try to understand how this simple CPU works, from the ground up. It is not a modern CPU with decades of optimization, but it has the same core functionality. So if you have ever wondered how a computer truly works, buckle up, we are going on a journey.

1. How to Read This

- If a diagram is hard to understand, click it and step through each frame one by one using the arrow keys, with the provided descriptions. This only works if you are reading on my website, bytecolony.computer.
- If you are reading this as a book (print, PDF, or ebook), animated diagrams show a few key frames, so transitions and minor moments may be cut. When something is cut, a link points to the full animation at bytecolony.computer.
- Don't try to "memorize" every layout. Focus on the mental models, concepts, and what part each piece has to play.
- If a section feels dense or hard to understand, follow the diagrams first and try to get a feel for what is happening.

2. Circuits & Switches

First, we need to know the basics of how electricity and circuits work. Here is a simple circuit:

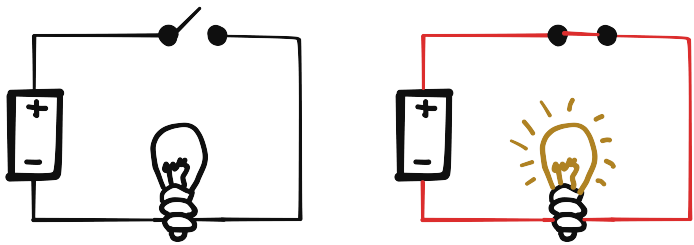


Diagram 2.1. A basic circuit with a battery, switch, and bulb.

A battery has two sides, + and -. On the - side, the atoms have an excess of electrons. On the + side, the atoms are starved of electrons. Voltage is how big that imbalance is: the difference in potential between the two sides. The bigger the difference, the harder the electrons are pushed and

pulled from - to +. A 9-volt battery pushes/pulls harder than a 1.5-volt one.

When we connect the two ends of a battery with a wire, the electrons are able to flow. That's because wires are made of metals like copper, which are conductors: their electrons can move easily from atom to atom. Materials like plastic, rubber and air are insulators: they don't let go of their electrons easily, so almost nothing flows through them. As the electrons travel from the - side of the battery to the + side of the battery, they pass through the filament of the light bulb, turning on the light. The filament is a very thin wire, so it's harder for the electrons to squeeze through. This is called resistance, and pushing current through it makes the filament heat up so much that it glows. The battery has chemicals that keep that imbalance of electrons going, to keep the current flowing.

The filament's resistance is actually important: it limits how much current can flow. If we connect + directly to - with just a wire, there's almost no resistance in the way, so a huge amount of current flows. The wire and the battery heat up fast, the battery drains, and things can melt or even catch fire. This is called a short circuit, and it is bad.

Even though technically electrons flow from - to +, most resources in the real world show current flowing from + to -. This is because the direction was decided (thanks, Benjamin Franklin!) before electrons were discovered, and it turns out the direction doesn't really matter (mostly). So we just stick to thinking of it as + to -, even though in reality the physical electrons flow from - to +. Also, later on, when I say a wire is driven high or low, that just means it is connected to + or -.

This is just the tip of the iceberg when it comes to electrical circuits. There is a lot more interesting stuff to learn, so if you are curious, check out *Lessons in Electric Circuits* in the Further Reading section at the end.

Either way, current can only flow when the loop is complete. A switch is simply a controlled break in the loop, allowing us to break and complete the loop whenever we want. When the switch is closed, the loop is complete and current flows. When it's open, the loop is broken and nothing flows.

In the circuit above (Diagram 2.1), the circuit is answering a simple question. Is the switch closed? If it is, the light turns on. If it's not, the light turns off. This might seem silly at first, but this is the simplest question a circuit can answer.

Let's see if we can combine more switches so the circuit can "answer" slightly more interesting questions. Say we want to build a simple dog washer circuit: a circuit that, based on some inputs, can tell us whether to wash our dog or not. Our simple circuit is going to use a light bulb being on to mean yes, wash the dog. Light bulb off means no, don't wash the dog. So we start with an extremely simple version with two switches. In this first version, the switches are directly inside the bulb circuit. The person using the circuit can open or close each switch to answer a yes/no question.

Let's say switch 1 represents `STINKY`: whether the dog is stinky or not. Switch 2 represents `OLD_WASH`: has it been more than 5 days since the last wash? The rules for our first circuit are:

If `STINKY AND OLD_WASH`, the bulb is on. Or in other words, if the dog is stinky and its last wash was over 5 days ago, then wash the dog.

Here is the circuit:

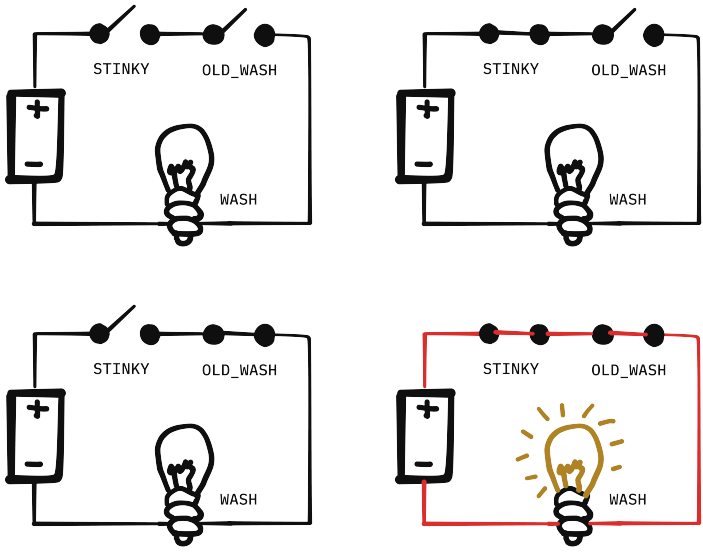


Diagram 2.2. The hand-switch version of AND.

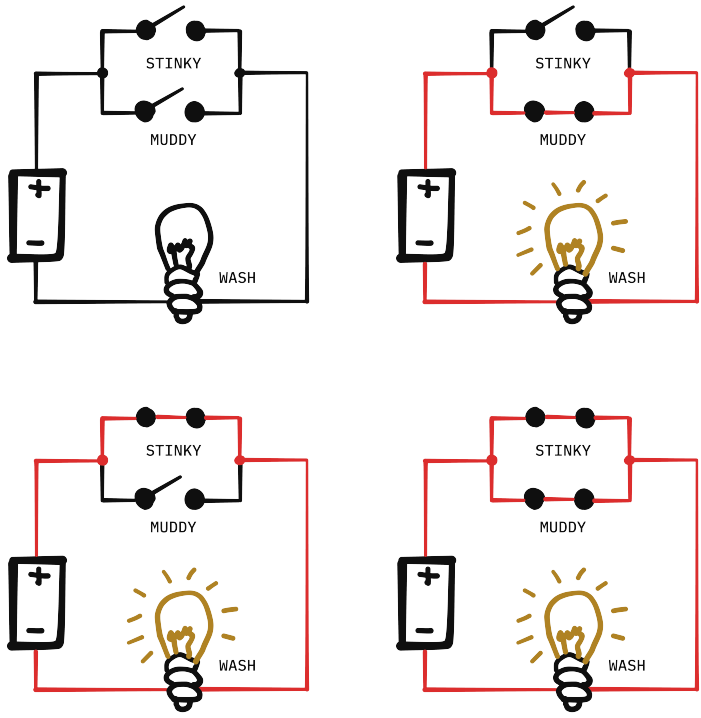
This circuit shows a logical AND operation. A person is flipping the switches manually. The output turns on only when both inputs are true.

Let's add a new input: `MUDDY`, if the dog is muddy. The rules of the circuit change:

If `(MUDDY OR STINKY) AND OLD_WASH`, the bulb is on.

All this says is, if the dog is muddy or stinky and it's been more than 5 days since the dog's last wash, you should wash the dog.

Let's focus on the `MUDDY OR STINKY` part of this circuit:



The full animation is at bytecolony.computer/writing/the-cpu/#f=or-gate-logical:1.

Diagram 2.3. The hand-switch version of OR.

This is a logical OR: at least one of `MUDDY` or `STINKY` needs to be on for the bulb to turn on. Both of them could also be on, and that is fine.

Now let's take a step back and talk about why we're doing this. What we just built are called logic gates: an AND gate and an OR gate. A computer is made of lots of these gates chained together, where the output of one gate becomes the input of the next. But right now, the inputs of our gates are switches flipped by a human, and the output is a bulb read by a human. To chain gates together, both the inputs and the output need to be wires, driven high or low. So how can we make a wire flip a switch?

3. Relays & Logic Gates

Electromagnetic relays, that's how. Or at least, that is one early solution to this problem. We will talk about other solutions a little more later on. This probably sounds quite complicated, but it is just a magnet powered by electricity.

> NOTE
> If you see many little batteries in a circuit, don't interpret that as several power sources. The battery symbols show if a wire is connected to power.

Here is how a relay works:

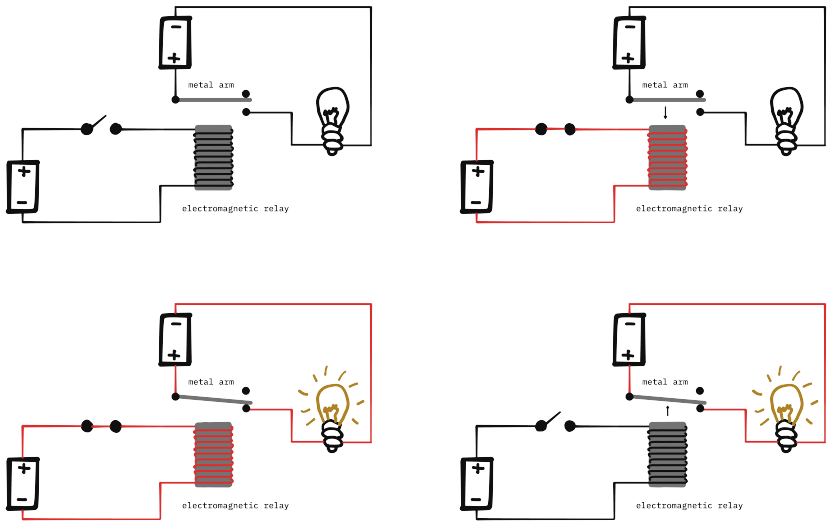


Diagram 3.1. An electromagnetic relay.

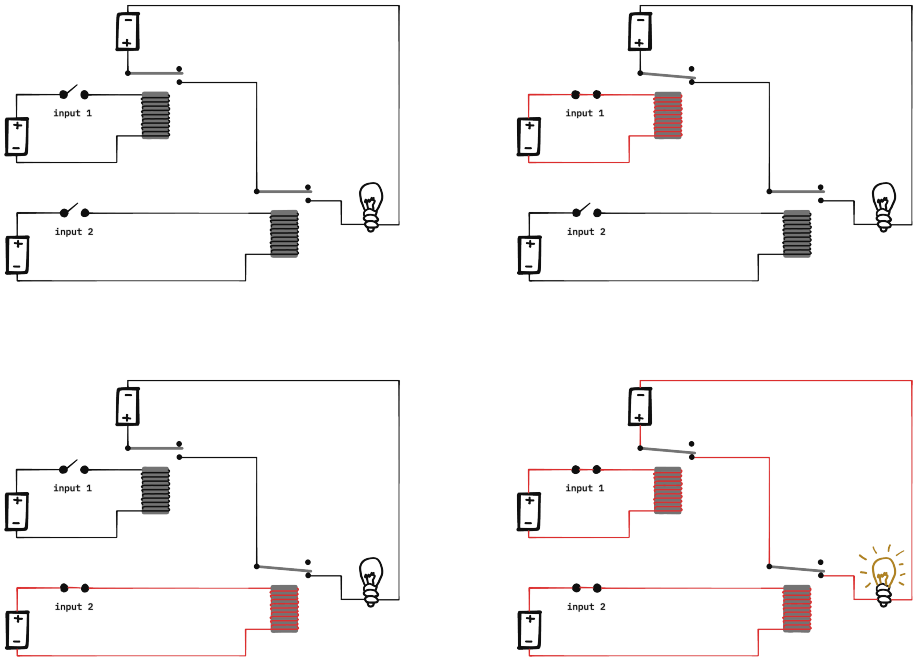
This relay is made from a coil of wire and a movable metal arm. When current flows through the coil, the coil becomes a magnet and pulls the arm down. When current stops, a spring pulls the arm back up. The coil is a long, thin wire wound up tight, so like the filament, it has enough resistance that it isn't a short circuit.

A relay lets one circuit open or close a switch in another circuit.

In this example, we end up using a switch in the input circuit anyway, but any kind of electrical signal could be used, like the output of another circuit. The switch is just there to demonstrate how the relay works.

As you can also tell by the diagram, there is a slight delay between the coil turning on and the metal arm moving. Relays are mechanical, so they do not switch instantly.

Now let's see how we can build an actual electrical AND gate that takes two input wires. For now, the output is still a bulb, but we will fix that soon.



The full animation is at
bytecolony.computer/writing/the-cpu/#f=electronic-and-gate:1.

Diagram 3.2. An AND gate.

The output circuit has two breaks in it, one controlled by each input relay. Only when both inputs are on do both relays close, completing the output loop.

That is the big difference from the hand-switch version. The inputs are still drawn as switches here, but they don't have to be. The input could just be a wire. One end of each coil is always connected to `-`, and the other end is the input wire. If the input wire is driven high, there is a voltage across the coil, so current flows through it and the relay pulls. If it is driven low, both ends of the coil are connected to `-`. There is no difference in potential between them, so the voltage is 0, no current flows, and the relay lets go. Using these relays chained in clever ways, you can create every fundamental logic gate, such as the OR gate.

But before the next diagram, I am going to use one more new symbol: ground. For our purposes here, the ground symbol will simply refer to the common return point of the circuit, usually connected to the negative side of the battery.

Every point marked with the ground symbol is connected together, as if there were hidden wires joining them underneath the drawing. It is not a new component. It is a less messy way to draw the return path of the circuit. The circuits are still loops. I am not explicitly drawing the return wire anymore.

> **NOTE**
> In a real schematic, the ground symbol itself would usually stay white. In these diagrams, I color it red when that return point is part of the active path for that frame. I think it makes the current path easier to follow visually, especially for beginners.

This is how the ground symbol looks:

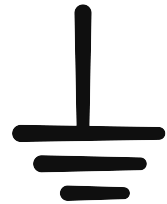
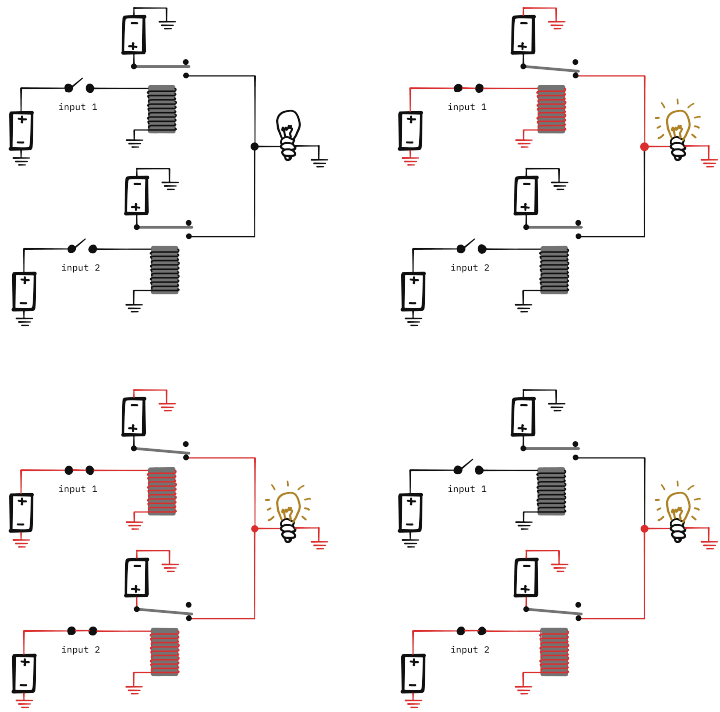


Diagram 3.3. The ground symbol.

Now here is the OR gate:

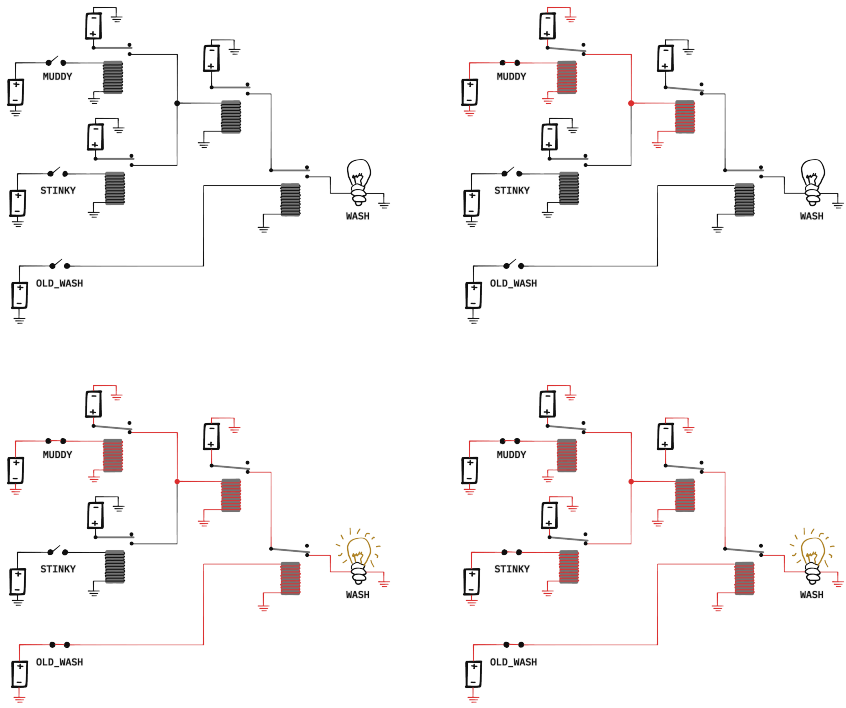


The full animation is at
bytecolony.computer/writing/the-cpu/#f=electronic-or-gate:1.

Diagram 3.4. An electronic OR gate.

That is an OR gate using relays. Instead of two breaks one after another like in the AND gate, the output loop has two paths side by side, one through each relay's arm. If either input is on, its relay closes its path, and the loop is complete. If both are on, the current just has two paths instead of one.

Now here is the full dog washer circuit up to this point:



The full animation is at bytecolony.computer/writing/the-cpu/#f=dog-washer-v1:1.

Diagram 3.5. The full dog washer circuit built with relays.

In a nutshell, if `MUDDY` or `STINKY` is on, and `OLD_WASH` is also on, the bulb turns on.

Notice the OR that checks `MUDDY OR STINKY`: its bulb is gone. Its output is now a wire, feeding straight into the next relay's coil. When `MUDDY` or `STINKY` pulls its arm, that wire gets connected to `+`, current flows through the coil, and the input relay for the next gate pulls. When

neither pulls, the wire is connected to nothing at all, so no current flows and the input relay for the next gate lets go.

That works, but from now on, our gates' outputs will mostly feed other gates' inputs (as we discussed earlier), so the output needs to be a wire, not a bulb. We also want that wire to always be in one of exactly two states: connected to `+`, or connected to `-`. Never disconnected (unless we are using a specific gate that we will cover later).

A `1` output is a wire being driven high. A `0` output is not "nothing"; it is a wire being driven low. Remember this information; it will come in handy later on.

Technically, with relays, we could get away with leaving a `0` output disconnected, like we have done here: an unpowered coil just stays off. We drive it low anyway. Real gates, built from transistors, need a `0` to actually be driven low for the logic to work correctly, and since we are using relays to replace them, we should still follow that rule.

All of our relay gates can be simply adapted to do this. As you can see in the "after" frames below, even when the relay is not pulling the arm, even when the output is `0`, it is still touching the negative end of that battery, so it is being driven to `-`.

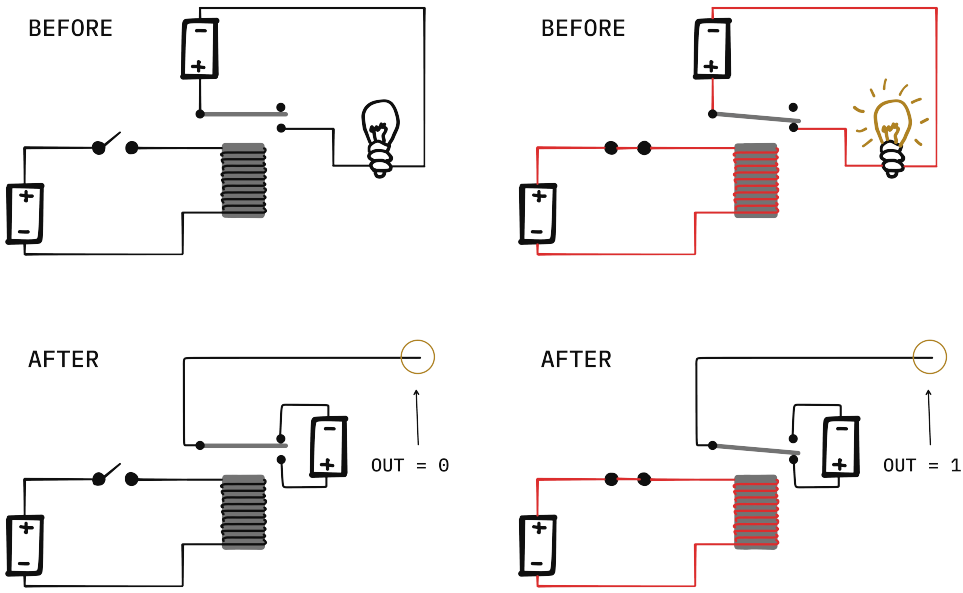


Diagram 3.6. Driving an output wire.

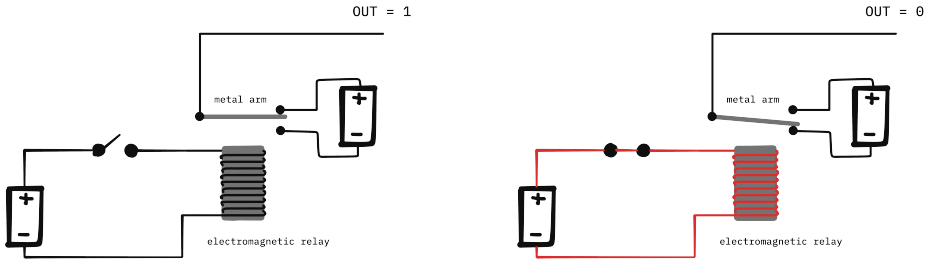
> NOTE
> In this diagram, a wire is only red when current is actively flowing through it, meaning it is part of a complete loop. That's why the output wire is still white when `OUT = 1`. It is being driven high, but it isn't part of a closed loop. It's just a single wire connected to `+`, so no current flows through it. Later, when we start drawing logic gate symbols and more high-level diagrams, we won't be working at the circuit level anymore, and red will just mean a wire is driven high, or `1`.

Okay, back to the dog washer. Let's introduce one last input, or "sensor": `RAIN_SOON`, whether it is predicted to rain soon. The rules of the circuit change once again:

```
((MUDDY OR STINKY) AND OLD_WASH) AND NOT RAIN_SOON
```

The parentheses indicate the groupings of logic, similar to math. So in plain English: if the dog is muddy or stinky and it's been more than 5 days since the dog's last wash and it's not going to rain soon, then wash the dog.

Let's focus on this NOT for a second. NOT just flips its input: if the input is on, the output is off, and if the input is off, the output is on.



The full animation is at bytecolony.computer/writing/the-cpu/#f=not-gate:1.

Diagram 3.7. A NOT gate.

Now before we look at the completed circuit, let's learn some basic logic gate symbols. An AND gate is drawn like this:

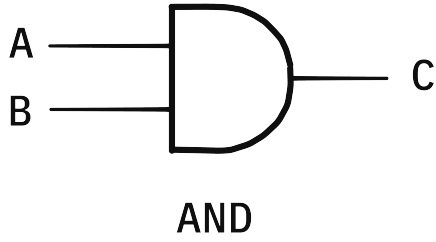


Diagram 3.8. An AND gate.

This symbol represents the AND circuit we made previously (Diagram 3.2), except instead of turning a bulb on and off, it drives an output wire.

An OR gate is drawn like this:

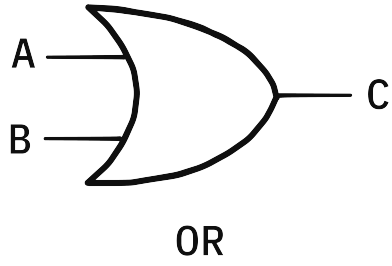


Diagram 3.9. An OR gate.

This symbol represents the OR circuit we made previously (Diagram 3.4). Whenever I use these symbols moving forward, they can almost directly translate to the circuits with the relays I showed you previously, just instead of turning on a light bulb, they drive a wire.

Here are three more useful gate symbols:

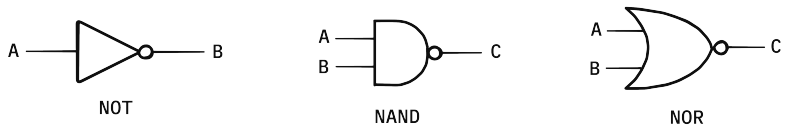
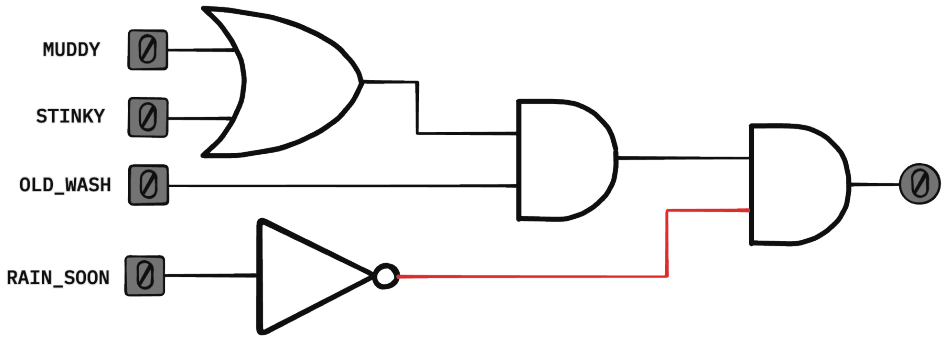


Diagram 3.10. NOT, NAND, NOR gates.

NAND is AND with the output flipped. NOR is OR with the output flipped. That little circle at the end of a gate means "flip the output."

> TRY IT
> A NAND gate gets `1` and `1`. What does it output?
> **Answer:** `0`. It's AND, flipped.

With our knowledge about logic gates, let's create the "should-I-wash-my-dog 5000" machine.



The full animation is at
bytecolony.computer/writing/the-cpu/#f=dog-washer-v2:1.

Diagram 3.11. The final dog washer circuit.

The electromagnetic relays (Diagram 3.1) we used in the examples are quite big and slow. Relays aren't the only solution. They are simply an early solution, and an easy one to understand, and many real computers like the Harvard Mark I actually used these types of relays.

In modern computers, similar behavior is achieved by using transistors. A transistor is kind of like a relay, but with no mechanical moving parts. A voltage on one pin decides whether current can pass between the other two. This makes transistors much faster and smaller.

This is how circuits make yes/no decisions. The circuit doesn't "understand" what `MUDDY` means. It just makes simple logical decisions.

A wire is just a wire. We gave these wires meaning. We decided that one wire means `STINKY`, another wire means `MUDDY`, and another means `RAIN_SOON`.

Addition seems like a pretty logical next step after these logic gates, but not so fast.

To make a CPU, we need to give wires a different kind of meaning: numbers. Before we can build a circuit that adds, we need a way to represent numbers using only on and off. That is what the next section is about.

4. Counting With Wires

First, let's define some terms. A wire driven low is `0`, and a wire driven high is `1`. Let's call one wire one bit. A bit can either be `0` or `1`. These are labels that represent the state of a wire.

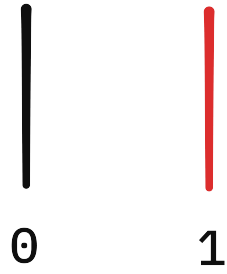


Diagram 4.1. One wire can represent two states: `0` or `1`.

If we want to represent numbers using wires, we are going to need more than one wire, because one wire can only represent up to two numbers, since it only has two possible states: `0` or `1`.

But two wires have `2^2`, or four states, and three wires have `2^3`, or eight states. That would allow us to represent more numbers.

Here are all the possible states we have with 3 wires:

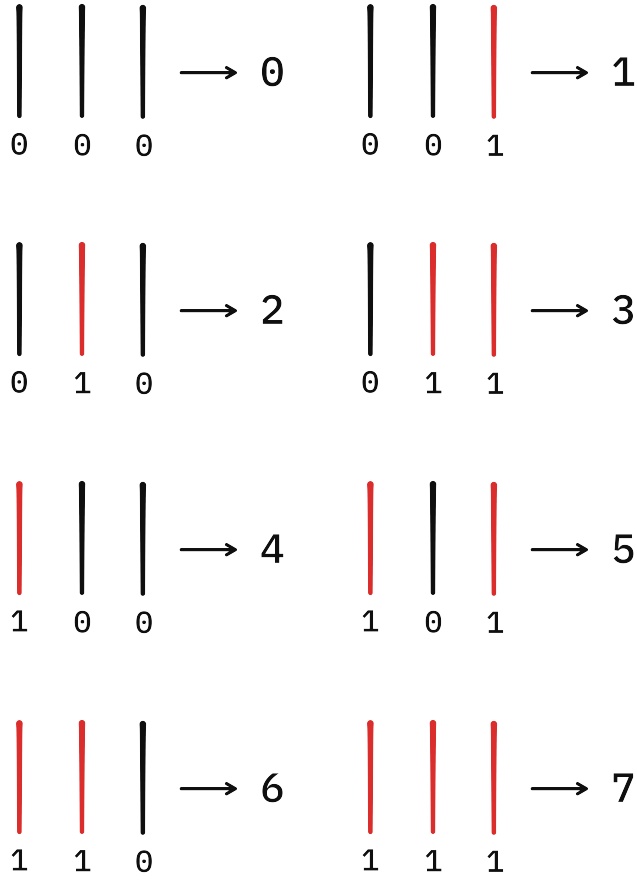


Diagram 4.2. States with 3 wires.

We can represent 8 numbers just like this. Keep going to 8 wires and there are `2^8`, or 256, possible patterns. A group of 8 bits is called a byte. If we use those patterns to represent non-negative numbers, one byte can represent 0 through 255.

But why does `010` mean 2? Why does `101` mean 5? Is it just randomly assigned? Not exactly. To understand this, let's take a quick detour to decimal, a.k.a. base ten.

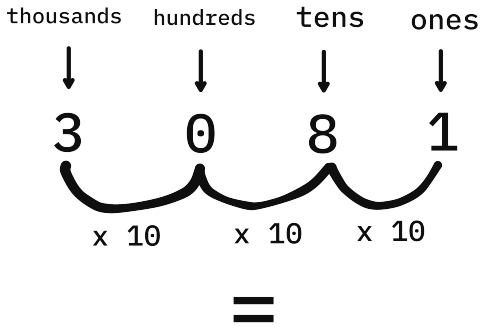


Diagram 4.3. The decimal system.

In our decimal counting system, each place-value is a power of 10. That is because we have ten digits: 0–9. This exact same place-value logic can apply to the binary system too. We have two digits, 0 and 1, so each place is a power of 2.

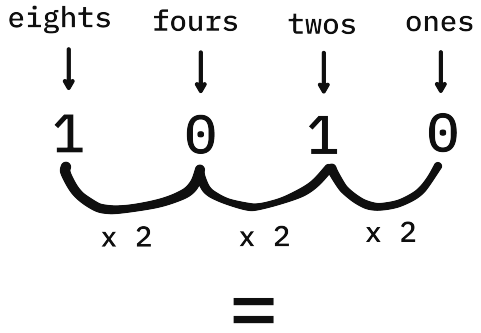


Diagram 4.4. The binary system.

Binary is, at the end of the day, the same place-value idea as decimal, just with two digits instead of ten. A few examples:

- `101` means 5
- `1101` means 13
- `101010` means 42
- `1100011` means 99

You don't need to do these problems in your head, but I hope the idea of how binary works makes sense. Let's walk through `1101` together.

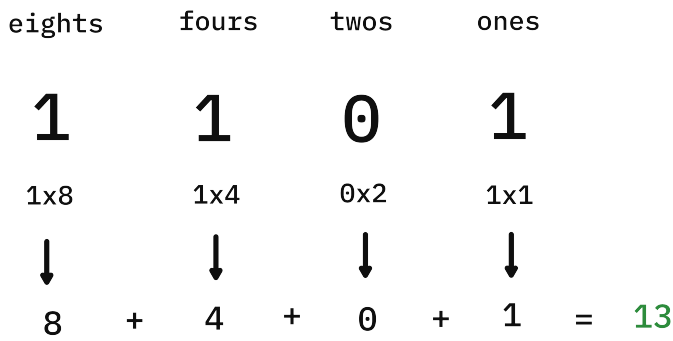


Diagram 4.5. An example in binary.

> TRY IT
> What is `00001010` in decimal?
> **Answer:** 10, which is 8 + 2.

So now that we can represent numbers with wires, how can we add numbers together? That is what the next section is all about.

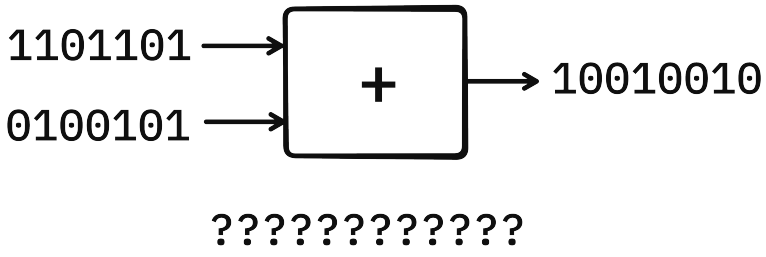


Diagram 4.6. Math?

5. Making 1 + 1 = 10

Let's start with a brief reminder of how we algorithmically add two decimal numbers.

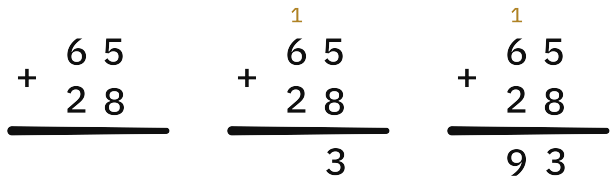


Diagram 5.1. Standard decimal addition.

We start at the rightmost column, do 5 + 8, get 13, and carry the 1. So we write 3 as the sum, and 1 as the carry. We then move left and repeat over and over, remembering to add any carry-in values. Binary addition works the same way.

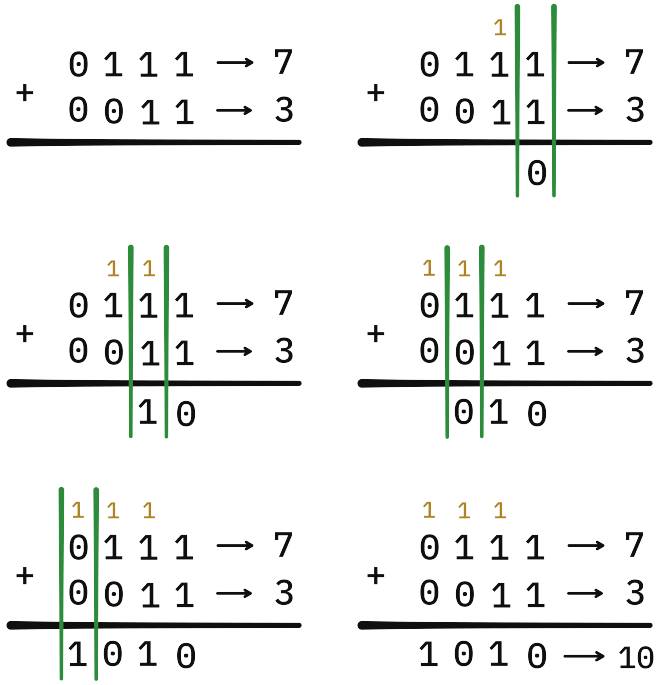


Diagram 5.2. Binary addition.

So yes, `1 + 1 = 10`. Not ten. `10` is binary for 2.

So the sum bit for that column is `0`, and the carry is `1`. `1 + 1 + 1` in the second column gives `11`, which is binary for 3. So the sum bit is `1`, and the carry is `1`, and so on.

How do we build a circuit using logic gates that performs this standard addition algorithm? Well, let's start with the rightmost column.

If we think about it, all the possible states are:

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

So just `0 + 0`, `1 + 0`, `1 + 1`, or `0 + 1`. If we can make a tiny circuit that takes two inputs, and produces two outputs that match these combinations, we have added the first column.

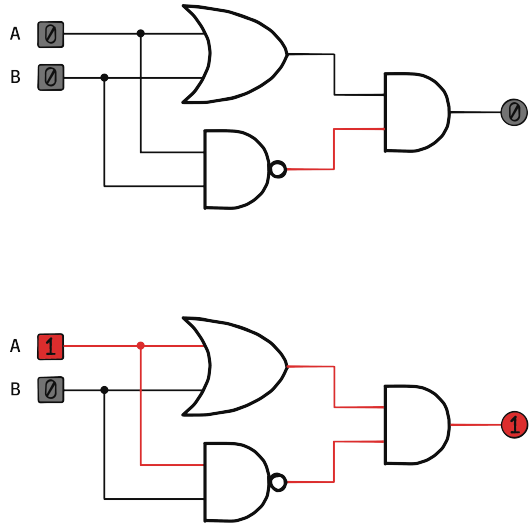
This is called a half adder. A half adder adds two bits, but it does not handle a carry-in value. That is the job of a full adder.

Let's first build this half adder. Let's start by computing the sum, not the carry-out. This is what we want our circuit to do:

A	B	Sum
0	0	0
0	1	1
1	0	1
1	1	0

The sum is `1` only when exactly one input is `1`, not both. This is called XOR, short for exclusive OR.

If we combine an OR gate and a NAND gate, and AND them together, we get XOR:



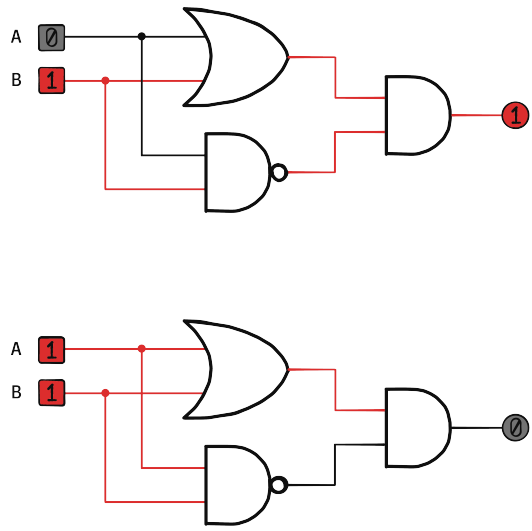


Diagram 5.3. Half adder sum / XOR.

OR checks that at least one input is on, and NAND makes sure the inputs aren't both on. Let's package that circuit into a gate called XOR.

Here is how an XOR gate looks:

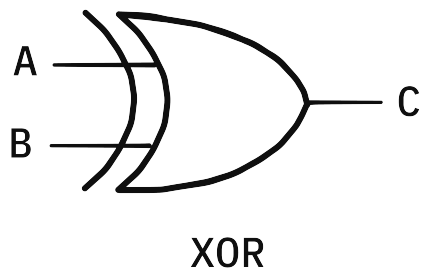


Diagram 5.4. An XOR gate.

Now let's do the carry value. The carry is simple. If you remember, for our half adder, the only time we want to carry is when we are doing `1 + 1`, so we use an AND gate to check if both inputs are on.

Here is our half adder:

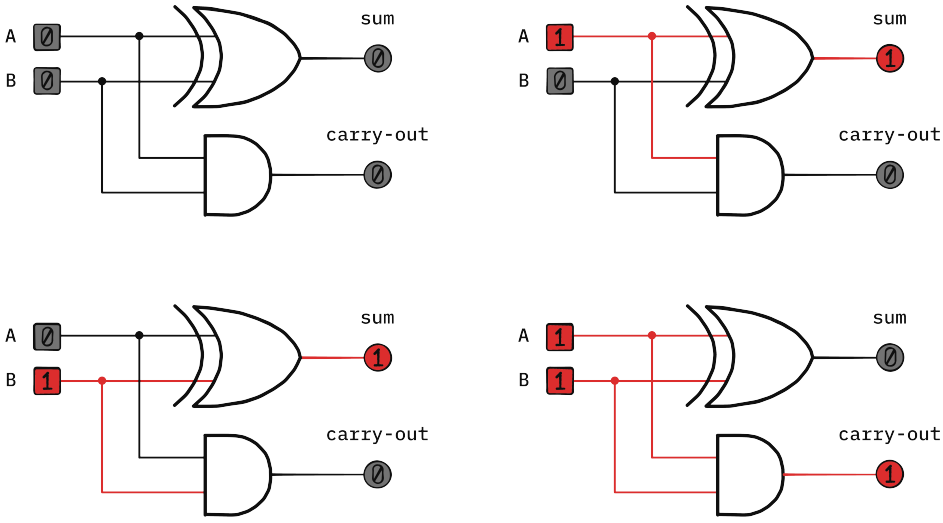


Diagram 5.5. A half adder.

As you can see, it works! `0 + 0 = 0`, `1 + 0 = 1`, `0 + 1 = 1`, and `1 + 1 = 10`.

> TRY IT
> What are the sum and carry for `1 + 1`?
> **Answer:** Sum `0`, carry `1`.

Now let's package up our half adder into a little box. From now on, I will call these packaged-up circuits "chips":

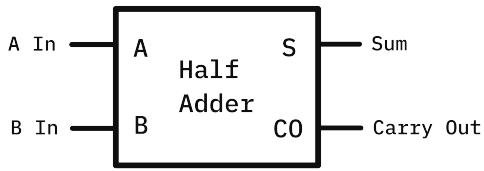


Diagram 5.6. A half adder chip.

Now that we have a half adder, we can add the rightmost column. That works because the rightmost column has no carry-in from a previous column. It only needs to add two bits.

So if we have a problem like this:

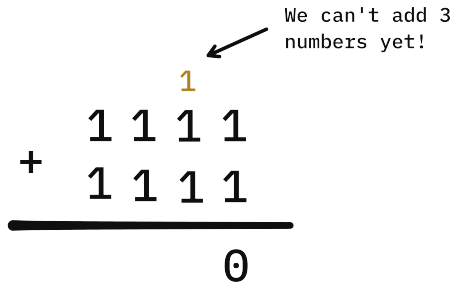


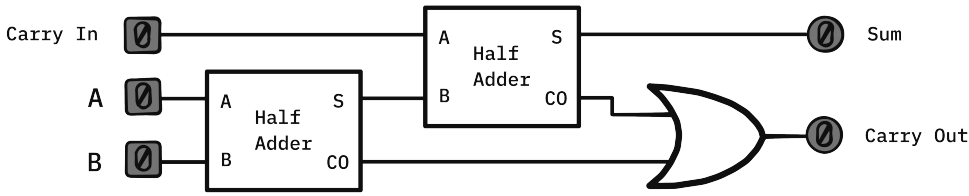
Diagram 5.7. The next column has to add two bits plus a carry-in.

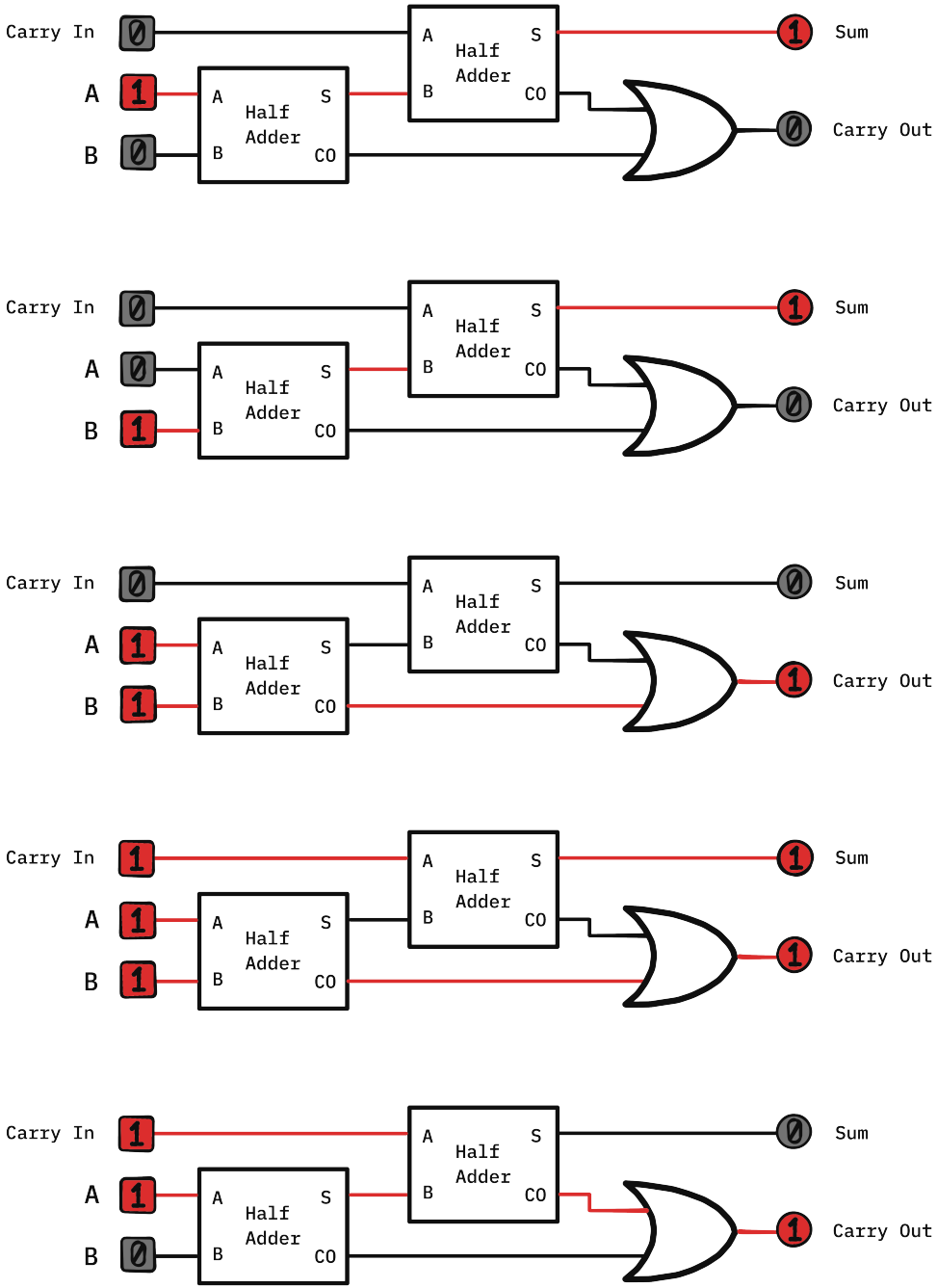
The half adder can handle the first column: `1 + 1`. That gives us a sum bit of `0` and a carry-out of `1`.

But now the next column has three things to add: `1 + 1 + 1`. The two original bits, plus the carry from the previous column.

A half adder cannot do that. It only accepts two inputs. To continue adding up the other columns, we need a circuit that can take in three inputs: `A`, `B`, and carry-in.

To add three bits, we use two half adders and an OR gate:




Carry In
A
B
Half Adder
A
B
S
CO
Sum
Carry Out

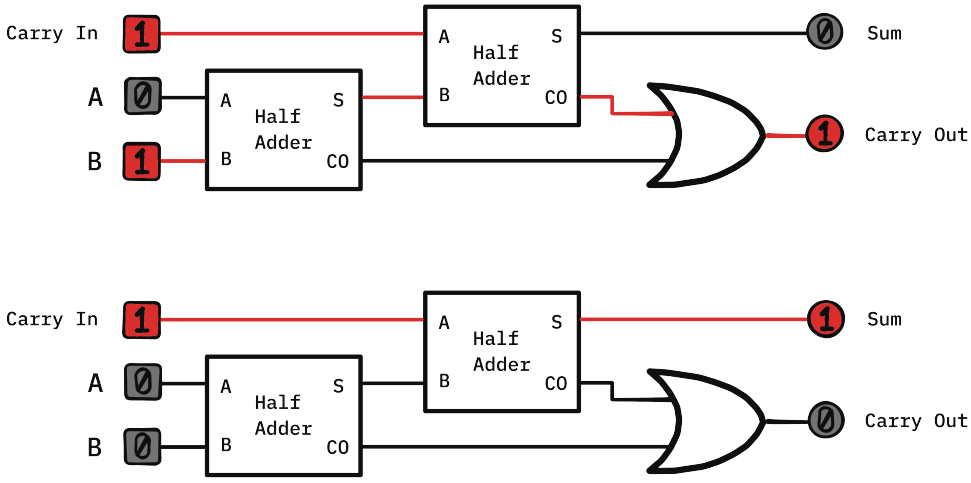


Diagram 5.8. A full adder.

This might look confusing at first. What if both half adders output a carry-out at the same time? That actually never happens. If the first half adder outputs a carry, its sum bit is `0`, and that `0` is one of the second half adder's inputs, so the second one can't carry. So both can never output carries at the same time. Take a moment to think about this if you are confused. So we can confidently OR the two carry outputs together. If either one is `1`, the full adder's carry-out is `1`.

Let's again package this up into a chip:

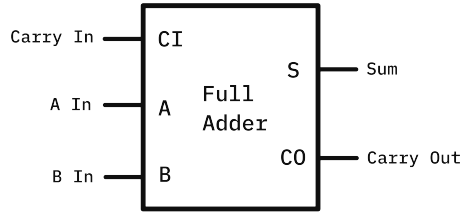


Diagram 5.9. A full adder chip.

We have made a full adder. Now we can chain full adders together to add two 8-bit numbers. Since 8 bits make one byte, this is an adder that can add two one-byte numbers: anything from 0 to 255.

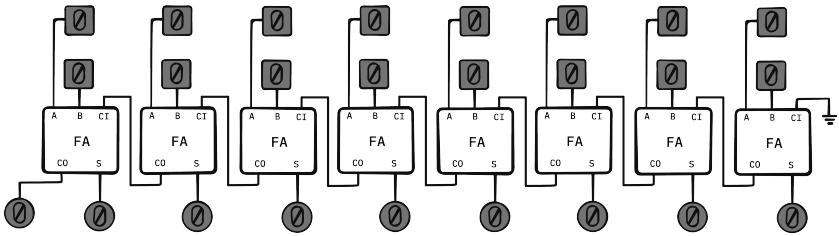


Diagram 5.10. An 8-bit adder.

Each full adder handles one column. The carry-out from one column becomes the carry-in for the next column. We did it, that's all addition is!

Keep in mind, the carry-in for the first adder is set to ground, a.k.a. 0. In the chip version below I bring that first carry-in out as a `CI` pin instead, so the chip can be chained too. For now we just feed it 0. Also, notice how we have 9 outputs, not 8. That is because two 8-bit values can add up to a number too big to fit in eight bits. It's like how adding two two-digit numbers could result in a three-digit number for us. Like 50 + 50 = 100.

Let's package this up into a chip once again:

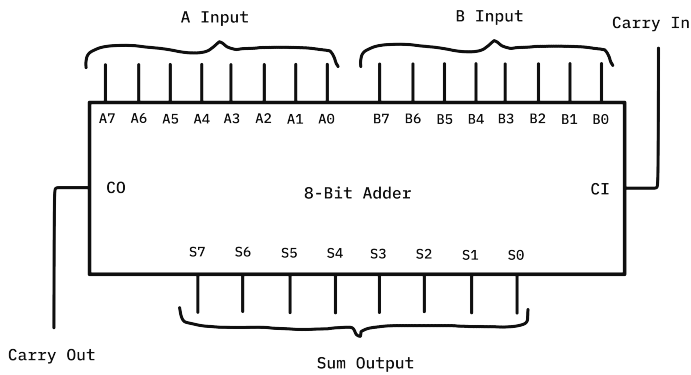


Diagram 5.11. An 8-bit adder chip.

Now we have the carry-out and carry-in as separate inputs and outputs and the whole adder nicely organized into this chip.

Let's have a look at some example problems:

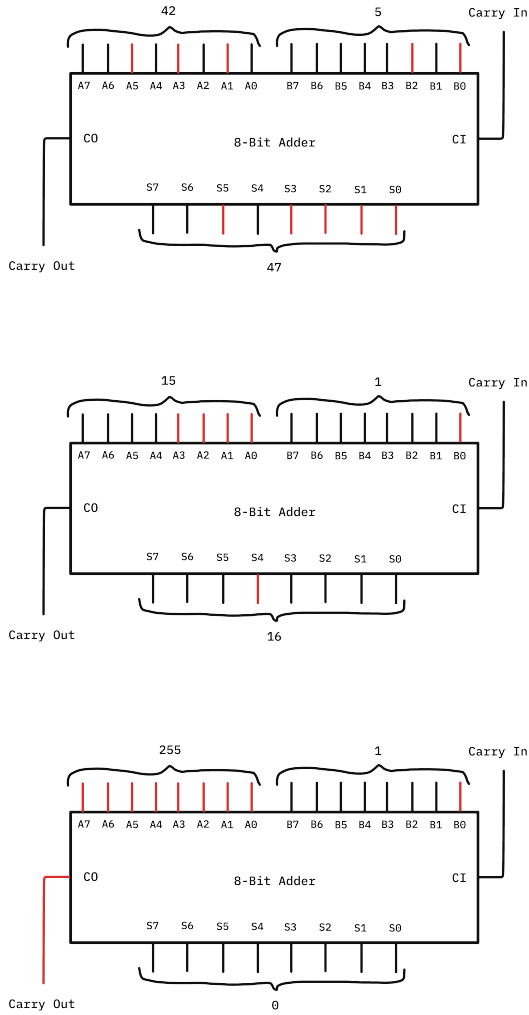


Diagram 5.12. Some examples on the adder.

As you can see in the third example, adding 1 to 255 turns every sum bit to `0` and turns the carry-out on.

This doesn't mean the adder got the wrong answer. In fact, 255 + 1 is `1 00000000` in binary: eight `0` output bits, plus one extra carry-out bit on the left. If we only look at the one-byte output, the result looks like `00000000`, or 0. If we also look at the carry-out, we can see that the real answer was 256.

That is called an overflow: the result was too large to fit inside one byte, so the extra information spilled out into the carry-out bit.

> TRY IT
> What does the adder output for `00000011 + 00000001`?
> **Answer:** `00000100`, which is 4.

With a bit of extra circuitry around the adder, we can also make little status wires, called flags. For example, if the answer is `00000000`, a `ZERO` flag can turn on. If addition spills past one byte, a `CARRY` flag can turn on. So `11111111 + 00000001` gives `00000000` with carry-out `1`.

I don't want to go deep into flags yet. Just remember that the adder can output little yes/no facts about the sum. That matters later for instructions like "jump if zero." But let's not get ahead of ourselves.

We have just built addition. But we also need something else: storage. For example, let's say we want to build a circuit that counts by ones, like 1, 2, 3, 4, …

The obvious idea is to feed the output of the adder back into one of its inputs. Start with `00000000`, add `00000001`, get `00000001`. Feed that back in, add `00000001` again, get `00000010`. Then `00000011`, then `00000100`, and so on.

That seems correct at first glance. But there is a big problem. An adder just looks at its current inputs and computes an output.

So if we wire the output straight back into the input, there is no stable value anymore.

The adder is basically being asked to make a number equal to itself plus one:

```
input = input + 1
```

That can never settle. As soon as the output changes, the input changes too, which means the output has to change again, which means the input changes again. With relays, you might physically see this mess play out. With transistors, it would happen almost instantly.

There is no boundary between the old value and the new value. There is no clean "step 1, step 2, step 3."

So this is not enough. We need a circuit that can hold a value, then update it only when we tell it to. That is the next problem: memory.

Part 2: Machinery

Before we start, here is what we have so far:

- A switch can complete or break a loop.
- Combinations of relays can make logic gates (AND, OR, NOT, and others).
- A row of wires can hold a number in binary, one bit per wire.
- Combinations of logic gates can be used to add two numbers.

6. Storing a Bit

In the last section, wiring the output of a circuit back into its input led to chaos. But with a smarter configuration, we can use this idea to store a bit.

The simplest circuit that stores a single bit is called an SR latch. SR stands for set-reset. The value `Q` is the output we really care about. If it is `1`, that means the latch is storing a `1`; if it is `0`, the latch is storing a `0`.

The diagram below also shows a second output written as a Q with a bar over it. That is just how engineers write `NOT Q`, pronounced "not Q". It always holds the opposite of `Q`. I will write it as `NOT Q` in the text.

The two inputs are `SET` and `RESET`, drawn as little buttons in the diagram: gray means not pressed, red means pressed. Pressing `SET` forces `Q` to `1` and pressing `RESET` forces `Q` to `0`. For this circuit to be used properly, `SET` and `RESET` should never be on at the same time.

The cool part is, if both `SET` and `RESET` are `0`, then `Q` is whatever we last did to it! The output loops back into the circuit, so the current state keeps reinforcing itself. This is the basic concept behind memory.

This diagram should help this make sense:

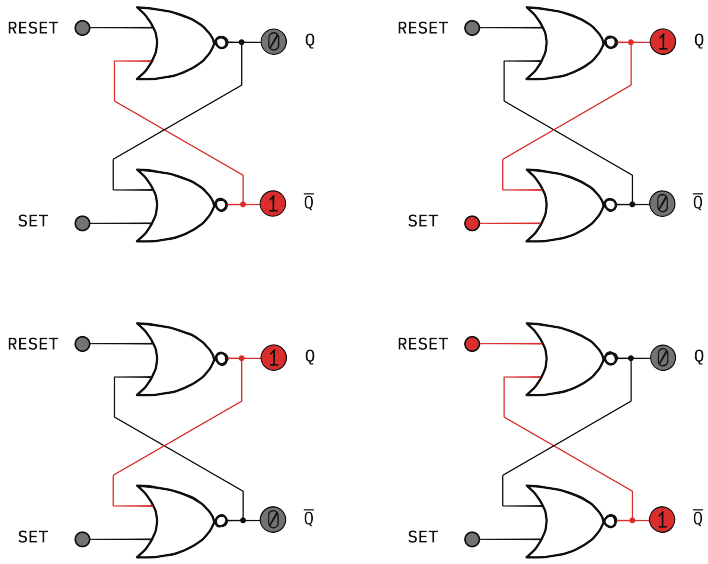


Diagram 6.1. An SR latch.

One quick reminder before we trace through this: a NOR gate only turns on when both of its inputs are off. Now let's trace through it.

If `SET` is on, the bottom NOR gate has to output `0`, because one of its inputs is on. That makes `NOT Q` equal to `0`.

Now the top NOR gate sees two `0` inputs: `RESET` is `0`, and `NOT Q` is `0`. So the top NOR gate outputs `1`, making `Q` equal to `1`.

Then even if we turn `SET` back off, the latch stays in that state. `Q` is still `1`, which keeps forcing `NOT Q` to `0`, and `NOT Q` being `0` allows `Q` to stay `1`.

`RESET` works the other way. If `RESET` is on, it forces `Q` to `0`, which allows `NOT Q` to become `1`. Then even after `RESET` turns off, `NOT Q` keeps forcing `Q` to stay `0`.

The circuit has state. Its output depends not only on the current input, but on what happened before.

Now that we have the core mechanism, let's refine the interface. Right now `SET` and `RESET` are super clunky. While they demonstrate the mechanism, what we would really like to have is two inputs.

- `Data` (`D`)
- `Enable` (`E`)

When the enable wire turns on, `Data` gets stored in `Q`. Or in other words, when we turn the enable wire on, `Q` mirrors `D`. Then when we turn `E` off, `Q` stays stable with whatever `D` was last.

This type of latch is called a D latch, D meaning data. It can be made using the SR latch and a few extra logic gates.

It basically checks: if `D` is true and `E` is true, `SET` is true, and if `D` is false and `E` is true, `RESET` is true. That's it, so let's not worry about the exact implementation.

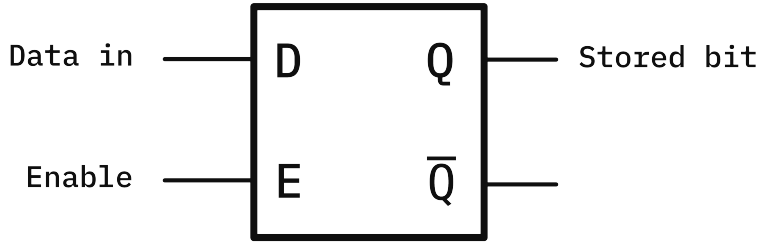
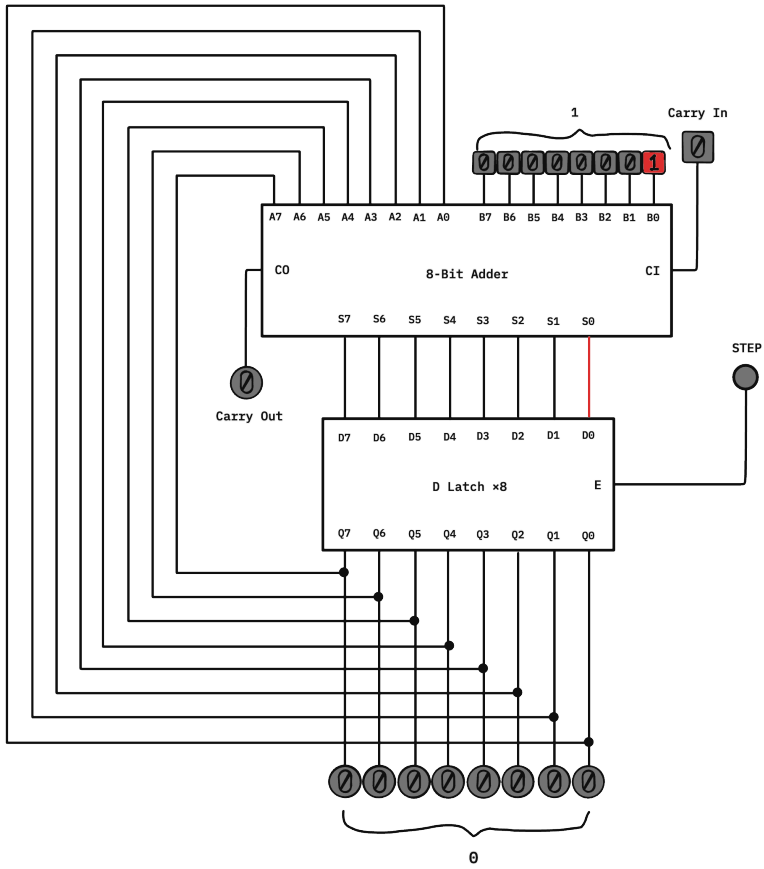


Diagram 6.2. D latch.

But we have a problem. Let's say we now try to use 8 of these D latches to hold the result from our adder, which would then feed back into one input of the adder (that counting circuit from the last section; I'll call it an accumulator from now on). It still wouldn't work.

Here is the problem: say we have the enable wire hooked up to a button. When that button is pressed down, the enable wire is on, so `Q=D` for that time. But if `Q` feeds back into the adder, and the result of the adder `D` changes quickly enough, `Q` can change again, jumping unpredictably based on how long we hold that button for.

If we want the accumulator to work correctly, we need the enable wire to turn on for an instant and then turn back off. That is just hard to do.



The full animation is at bytecolony.computer/writing/the-cpu/#f=d-latch-accumulator:1.

Diagram 6.3. D latch accumulator.

As you can see in this diagram, even pressing the button quickly jumps the result up by 5. With real transistors, even if you try to physically tap the button, it could count up by millions, overflowing these 8 bits thousands of times. How long you hold the button decides the answer. It doesn't count in ones.

But what if we had a storage circuit that only copied `D` into `Q` at the exact instant `E` turns on?

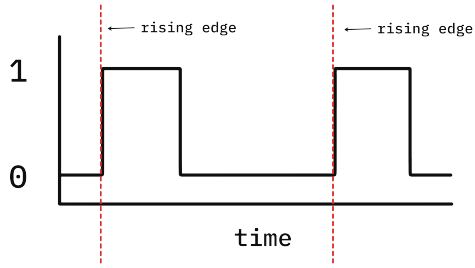


Diagram 6.4. The rising edge of a signal.

This graph shows the state of a wire. When the line is at the top, it is on. When it is at the bottom, it is off.

When a switch is flicked or a button is pressed, a transition happens. That transition is called an "edge", and when the wire turns from off to on, it is a rising edge.

Now what if we only set `Q` to `D` on that transition, at the rising edge? The edge is an instant of time, not a duration.

The circuit that does this is called a D-type edge-triggered flip-flop. This might sound like a mouthful, but D-type just means it takes in a data input, edge-triggered means it triggers on the edge of a signal, and flip-flop means it is a storage circuit similar to a latch, but usually edge-triggered.

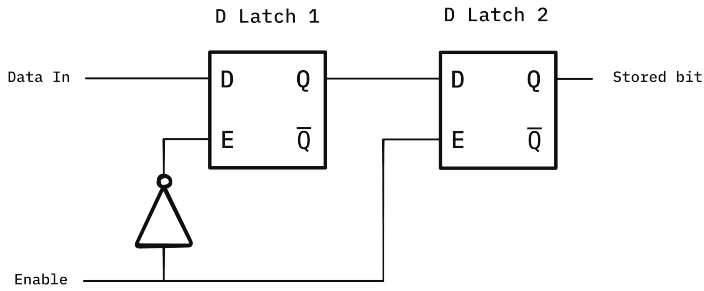


Diagram 6.5. A flip-flop.

How it works is, when the enable wire is off, the first latch mirrors `D`. That is because the NOT gate flips the enable signal, so the first latch sees it as on.

Then when enable turns on, the second latch stores the output of the first one. And because enable is now on, the first latch is locked, so it can't change.

So if `D` changes while enable is off, we are all good because the second latch is locked. But if `D` changes while enable is on, we are fine because the first latch is locked. So `Q` can only change at the exact moment `E` turns on. That is our rising edge.

Here is one storage cell, which is just the flip-flop we showed above:

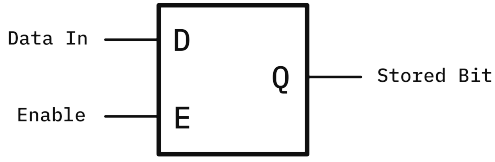


Diagram 6.6. A one-bit storage cell.

If we connect 8 of them side by side, we get one byte of storage:

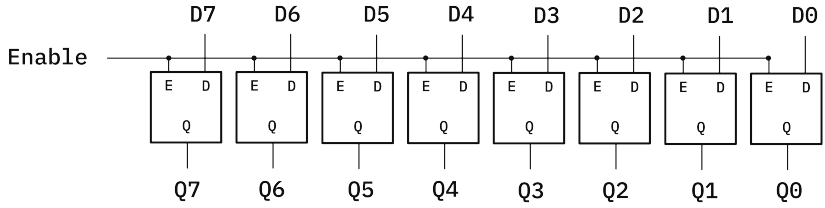


Diagram 6.7. Eight storage cells.

And we can put all that into a chip called an 8-bit register:

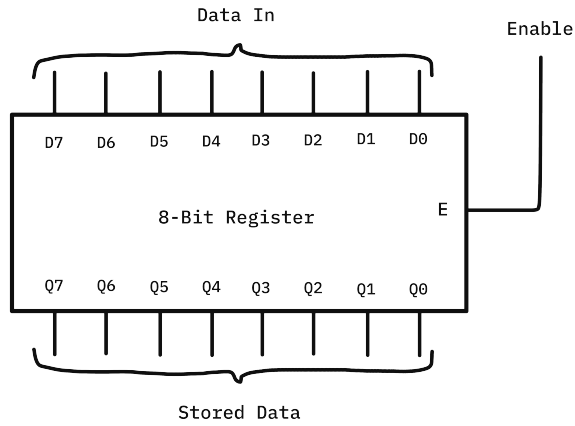
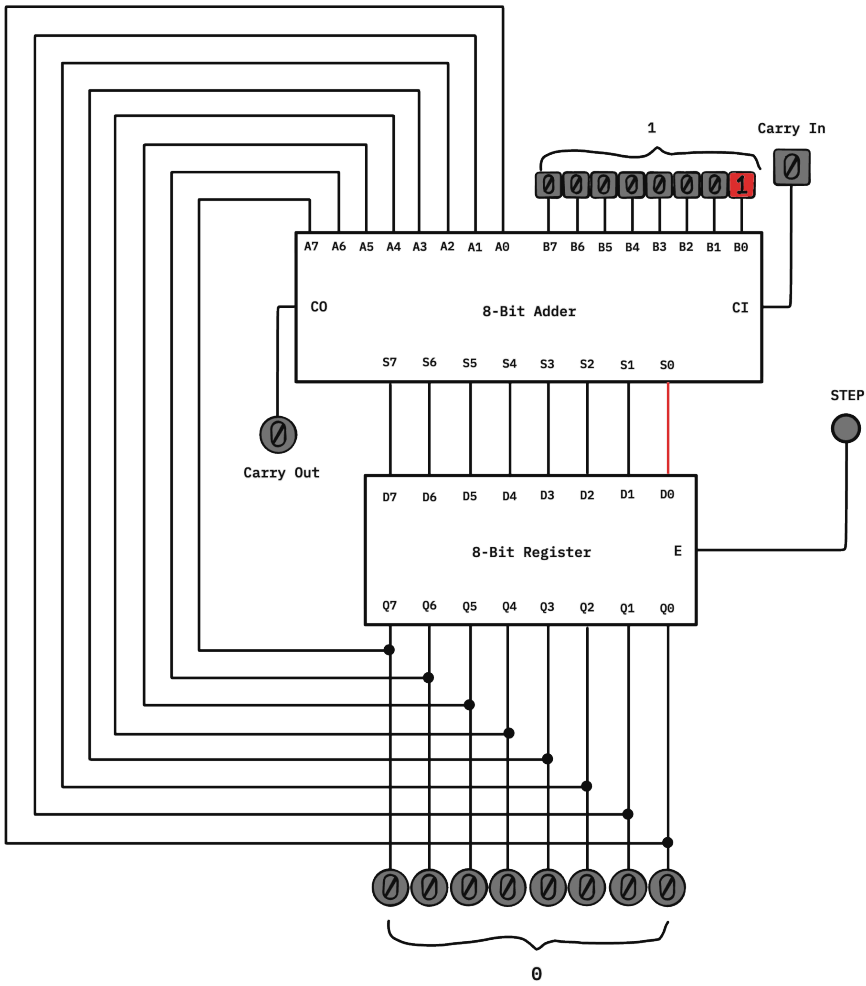


Diagram 6.8. An 8-bit register.

Now with this register, let's build a basic accumulator/adder circuit.



The full animation is at
bytecolony.computer/writing/the-cpu/#f=full-accumulator:1.

Diagram 6.9. Our full accumulator.

As you can see, the circuit kindly waits for us, and is incrementing by ones! How this works is, when the STEP button is pressed, the output from the adder gets saved into the register on the rising edge of that press. This then changes the input to the adder, which changes its output,

but the register holds its value because it only captures on the edge of the press. Holding STEP down does nothing special. So each press increments the register's value by, in this case, 1.

Now, a real computer would need to do these kinds of things millions or billions of times per second, and we don't have some human clicking a step button. What we have is a circuit that automatically goes on, off, on, off billions of times per second. This is called a clock. Each rising edge of the clock acts like one press of STEP.

Here is the basic concept of a clock:

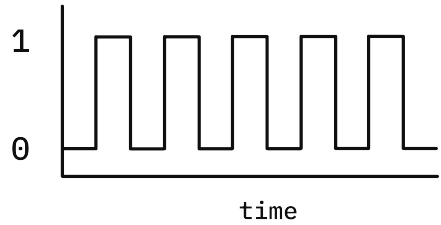


Diagram 6.10. A clock signal.

This repeating on-off behavior can be achieved in different ways. A rough toy example is feeding the output of a NOT gate back into its input, so the signal keeps trying to flip back and forth between on and off.

Real clocks are built in more sophisticated and reliable ways, often using crystals or other oscillator circuits. But we do not need to build the clock itself here. For now, we can treat it as a little chip that repeatedly produces the same on-off signal.

Just imagine the new accumulator with a clock signal instead of a STEP button.

We now have some storage. A register that can hold a byte, and update exactly when we want. But registers on their own are not enough. We need to be able to move numbers between registers, the adders, and the main memory, which we will build later.

7. Taking the Bus

One simple solution to move bytes around would be to give every component its own bundle of 8 wires to every other component, but that would become a mess very quickly.

A simpler solution is to have one single 8-bit data highway, where components can put data on and take data off. This collection of 8 wires is called a bus. I hope the title of the section now makes sense for any of you confused readers, sorry not sorry.

One more thing: moving forward, when I want to draw a collection of 8 wires, instead of drawing each wire, I will just draw a thick arrow that represents 8 wires.

To show the state of the wires, I can write a number in the arrow; the number 0, for example, means the wires are all off, and the number 2 would mean the wires are `00000010`, which is 2 in binary.

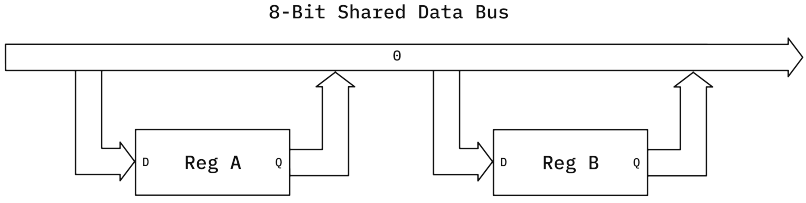


Diagram 7.1. Two registers sharing a bus.

But we have an issue: this diagram is technically not possible yet. Say register A is outputting `00000000` and register B is outputting `00000001`, both onto the same 8 wires. Notice what would happen to the last wire. Register A is driving it low, so that wire is connected to `-`. Register B is driving the same wire high, so it is also connected to `+`. What happens?

Yep. A short-circuit.

We need a way to connect these registers to the bus, but also let them get out of the way when they are not supposed to actively drive a wire to `-` or `+`, like I discussed previously (the "after" frames of Diagram 3.6).

Just setting the output wires to `00000000` is not enough. On a shared bus, `00000000` is not nothing. It is actively driving the bus to `-`. Of course, in our relay world technically we don't need to drive to `-` like I mentioned before, but in the real world, where transistors are used, we certainly do, and that causes problems.

One clean way to solve this problem is by using something called a tri-state buffer. It has two inputs, `E` and `D`, which stand for enable and data.

We have seen `E` before, in the context of "enable writing", but now we are using `E` in the context of "enable outputting".

If `E` is on, the output will just be whatever `D` is, so either `0` or `1`. If `E` is off, no matter the value of `D`, the output will be `Z`.

`Z` means the buffer's output is disconnected. Engineers call this "high impedance", which is where the letter `Z` comes from. It is not driving the bus to `+` or `-`, so another component can safely drive the bus without a clash or short-circuit.

Or in other words, this buffer is not touching the wire. `0` is very different: the component is actively driving the wire to `-`.

So we have three states:

```
1: driven high
0: driven low
Z: disconnected
```

This relay diagram of how a tri-state buffer works should make this concept crystal clear.

> NOTE
> I have drawn everything the output wire is currently touching in yellow. Yellow is just there so you can follow the path; it doesn't mean anything.

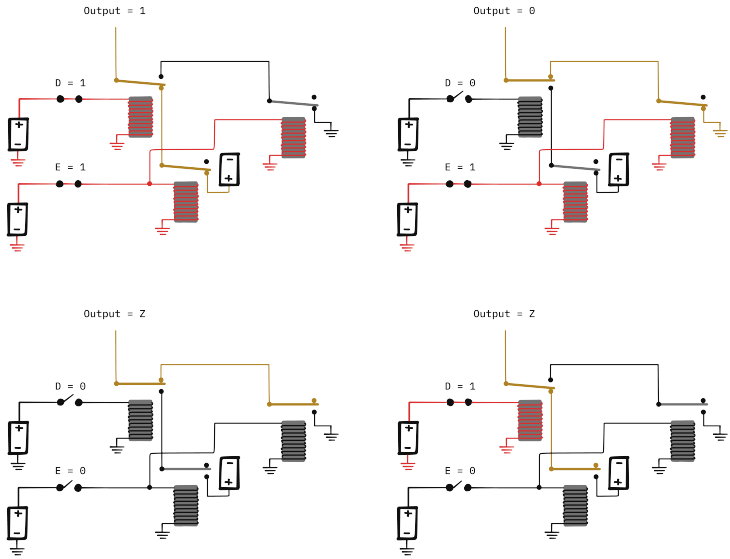


Diagram 7.2. A tri-state buffer built with relays.

This looks complicated, so let me break it down. First, ignore the two relays on the right and look only at the `D` relay at the top. Its arm is attached to the output wire, and it works just like the output driver from before (the "after" frames of Diagram 3.6). When `D` is `1`, the arm is pulled down onto the wire that leads toward the battery, `+`. When `D` is `0`, the arm goes up onto the wire that leads toward ground. Remember, ground is just the `-` side.

The important idea is that neither of those wires is directly connected to `+` or `-`. Each one has a relay in between. Both of those relays are controlled by `E`.

When `E` is `1`, both gaps close. The output is now connected to whichever side `D` picked, so it is driven to `1` or `0`.

When `E` is `0`, both relays touch a point connected to nothing. Both wires lead to a dead end. The output wire is touching nothing. That is `Z`.

Here is the full table:

E	D	Output
1	1	1
1	0	0
0	0	Z
0	1	Z

This is the logic gate symbol for a tri-state buffer:

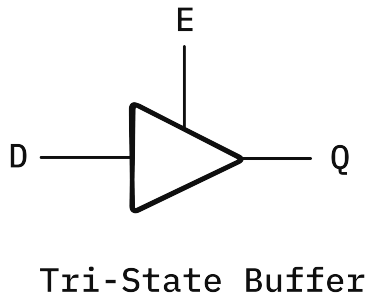


Diagram 7.3. A tri-state buffer logic gate.

> TRY IT
> `E` is `0` and `D` is `1`. What is the output? Is that the same as `0`?
> **Answer:** `Z`. It is not the same as `0`: `0` is actively driven low, but `Z` is not touching the wire at all.

Let's address this enable conundrum. We have two uses for the word enable, with completely different meanings and contexts. One means enabling writing, and the other means enabling output. From now on, we will use two separate terms to avoid confusion: `WRITE` and `OUT`.

We can make a new type of register, one with `WRITE`, `OUT`, a `D` input, and a `Q` output that shows the stored bits 0–7. By `D`, I simply mean the input data that we can store when `WRITE` is enabled.

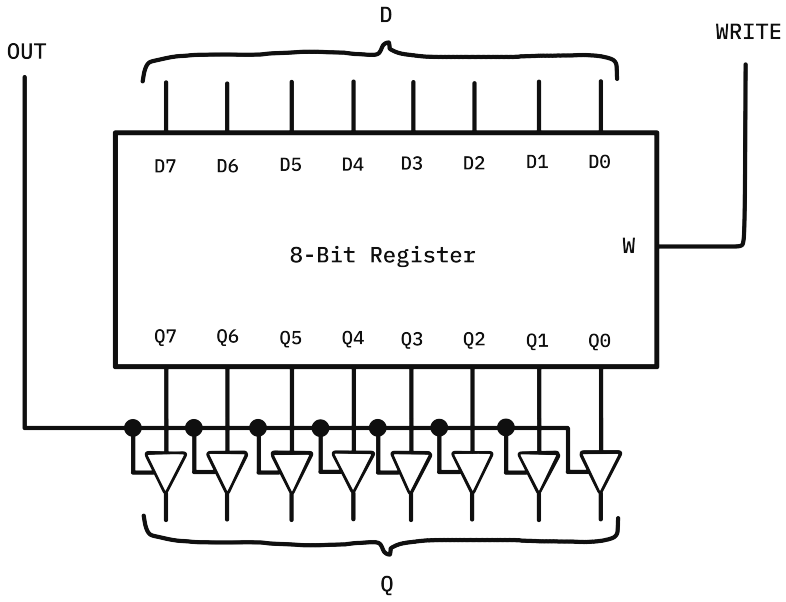
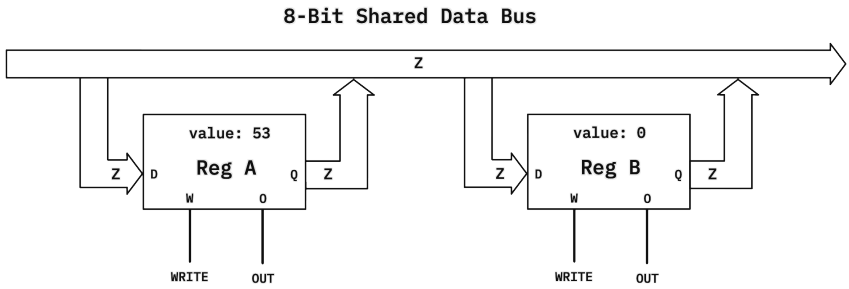
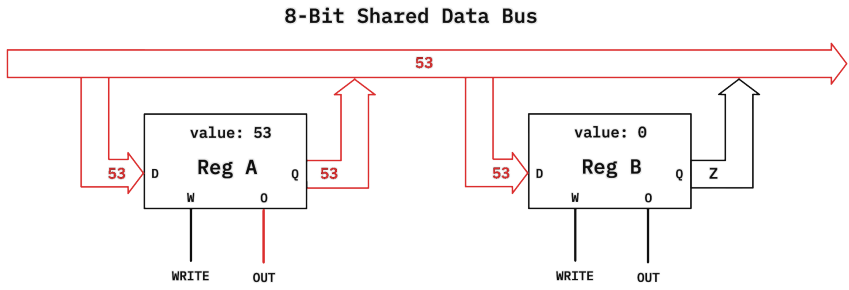


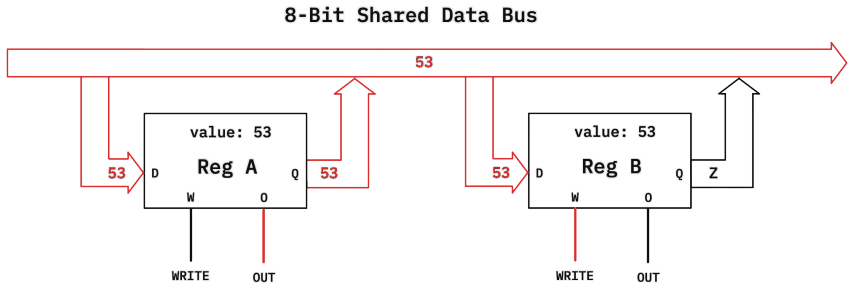
Diagram 7.4. Our register with an `OUT` input.

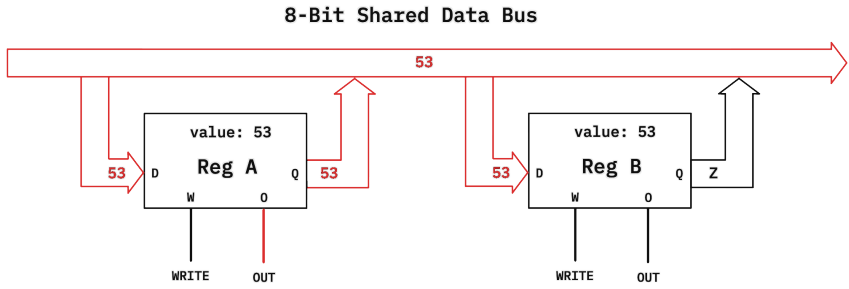
We are just connecting `OUT` to all of the enables in the tri-state buffers. So if `OUT` is `0`, `Q` will be all `Z`, and if `OUT` is `1`, `Q` will be whatever is stored in the register.

With that, we can use these new registers with a common bus to move data. Here is an example where the content of register A gets copied into register B.




8-Bit Shared Data Bus
53
value: 53
Reg A
D
Q
W
O
53
53
WRITE
OUT
value: 0
Reg B
D
Q
W
O
53
Z
WRITE
OUT


8-Bit Shared Data Bus
53
value: 53
Reg A
D
Q
W
O
53
53
WRITE
OUT
value: 53
Reg B
D
Q
W
O
53
Z
WRITE
OUT


8-Bit Shared Data Bus
53
value: 53
Reg A
D
Q
W
O
53
53
WRITE
OUT
value: 53
Reg B
D
Q
W
O
53
Z
WRITE
OUT

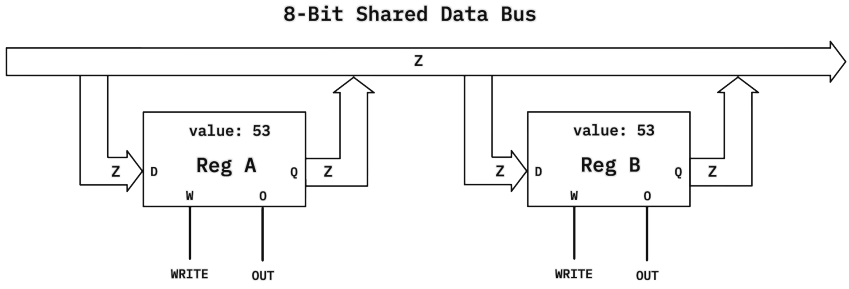


Diagram 7.5. Copying register A into register B through the shared bus.

I have some text inside the register that shows what it is storing. We have `W` and `O`, as well as `D` and `Q`, which are the inputs and outputs.

Of course, on the second frame, when `OUT` of register A is enabled, the `D` wires of both registers are also going to be 53 because they are directly connected to the bus.

Also generally in this diagram, register B's output is sometimes shown as `Z` even when the bus is 53. That is because register B's `OUT` is off, so register B is not driving the bus. It may be connected to a bus currently at 53, but the 53 is coming from register A. `Z` describes what register B is doing, which is nothing.

By the end of this sequence, we have copied the value 53 to register B. We can have many more registers and other components sharing a common bus, as long as only one of them is driving the bus at a time.

Now we can store a byte, compute a sum, and move bytes around. The next problem is organization and scale. How do we organize many stored bytes so the machine can choose one slot, read it, and write back to it? A handful of registers aren't enough.

8. Organizing Data

We want to build a system that organizes data into a simple structure.

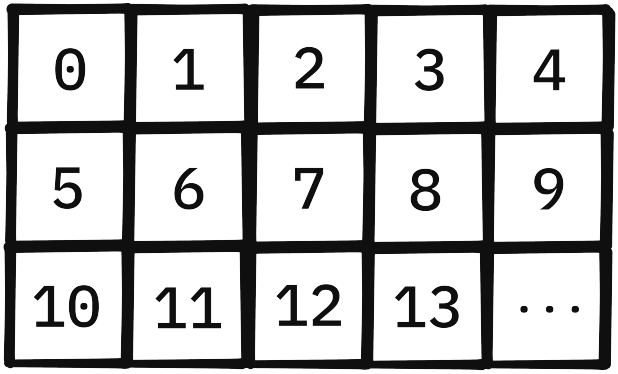


Diagram 8.1. Our data structure.

It is a set of slots, each with its own address. This system is known technically as RAM: Random Access Memory. It is called RAM because when the CPU wants to access a slot, it just knows the number and can access any slot at will. It is not like flipping through a book looking for the right page. It is more like grabbing a book from a bookshelf, where you already know exactly where the book is.

Now let's think about exactly what we would want this RAM chip to do, and what inputs and outputs it would have.

- `address`: the slot we wish to access
- `WRITE`: whether we want to write a value to this address
- `OUT`: whether we want to output the value onto the bus
- `data in`: the value we would like to write
- `data out`: the data output line

To be clear, `WRITE` and `OUT` are control signals, so 1 input wire each. For this demo RAM, `address` is only 4 input wires. `data in` and `data out` carry bytes and are both connected directly to the common

bus. This only works if no other part is driving the bus when `OUT` is enabled.

So we are going to build a minuscule 16-byte RAM: 16 addresses, with each address storing one byte. This design can be scaled up easily. Our address will be 4 bits long, because `2^4` is 16.

We could do this as a tall stack of 16 registers, but that would take 16 select wires, one for every slot. If we arrange the slots into 4 rows and 4 columns instead, we only need 8 select wires: 4 to pick the row and 4 to pick the column. It is just more efficient, and on bigger chips the savings get huge. So we will split the 4-bit address in half.

The bottom two bits pick the row, and the top two bits pick the column:

```
top 2 bits = column
bottom 2 bits = row
```

Two bits can choose 4 values, so this gives us a 4×4 grid of memory slots. That is 16 total bytes.

Once the address selects a slot, two things can happen:

- If `WRITE` turns on, the selected slot stores `data in`.
- If `OUT` is on, the selected slot drives its stored byte onto `data out`.

Let's start with building a simple decoder. This decoder will take 2 bits of our address and, based on that number, turn on exactly one of its 4 outputs.

In the diagram, the top bit is the bigger bit (the 2's place) and the bottom bit is the smaller bit (the 1's place).

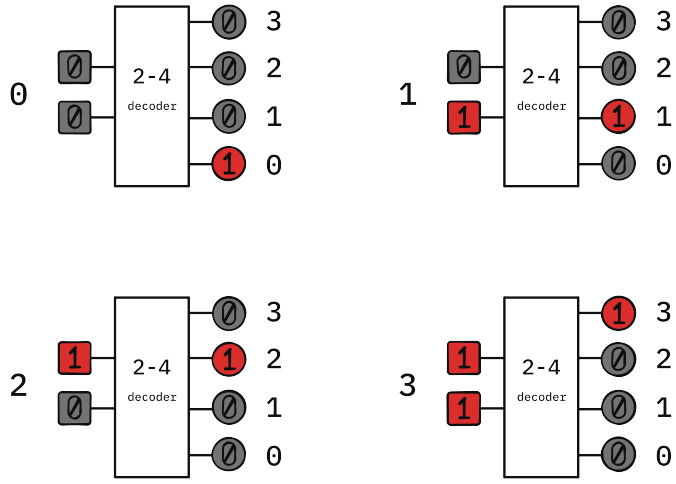


Diagram 8.2. How a decoder works.

As you can tell, no matter the inputs, exactly one output wire is on at a time. How a decoder works is extremely simple. It uses a bunch of logic gates to ask these simple questions.

- If `00` → turn on output 0
- If `01` → turn on output 1
- If `10` → turn on output 2
- If `11` → turn on output 3

Here is how it works if you care:

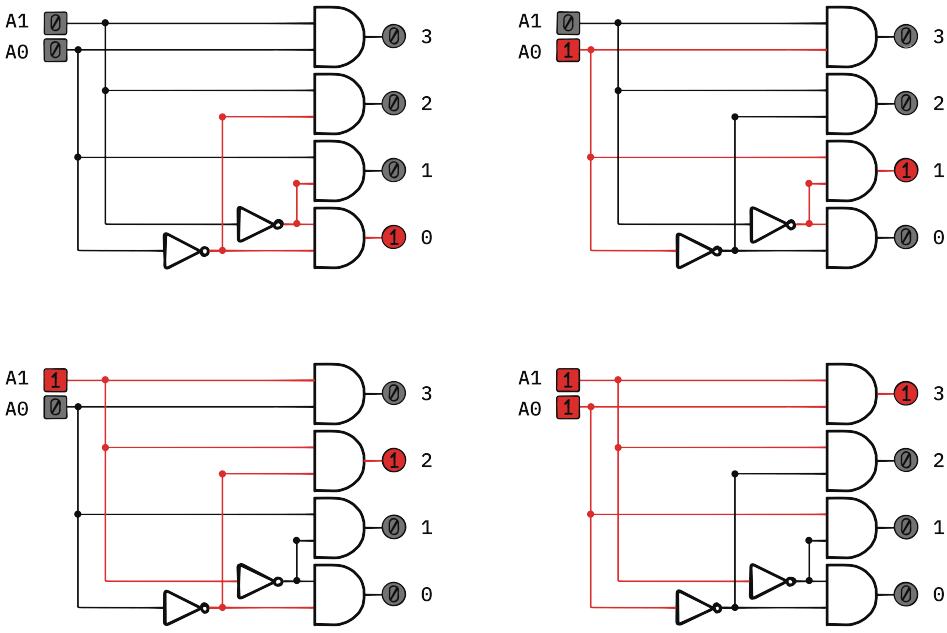


Diagram 8.3. 2-4 decoder internals.

We use one `2-4` decoder for the rows and another `2-4` decoder for the columns. Where the selected row and selected column cross, that is the byte we want to target. This diagram shows a few addresses as examples. Each address gets its own little intersection.

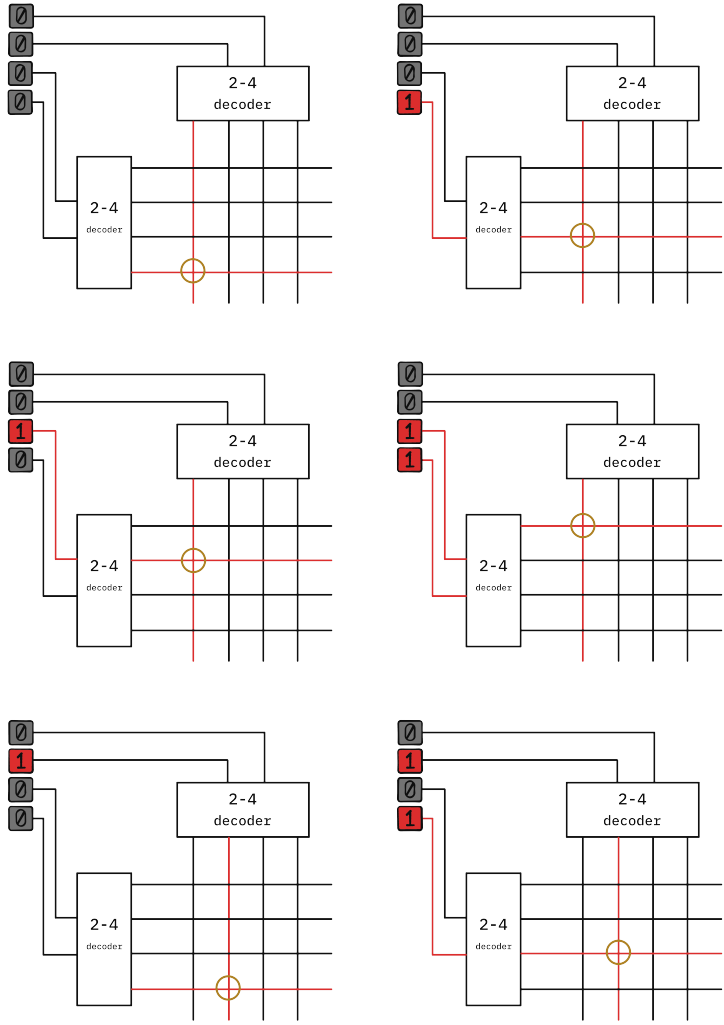


Diagram 8.4. Where the row and column meet.

Honestly? That's it. Two decoders, sixteen registers, some wires and buses, a few extra logic gates, and we have RAM.

> NOTE
>
> In the next diagram, blue lines are 8-bit data buses. `OUT` is yellow, and `WRITE` is orange. They are still ordinary wires; the colors are only there to make the diagram easier to follow.

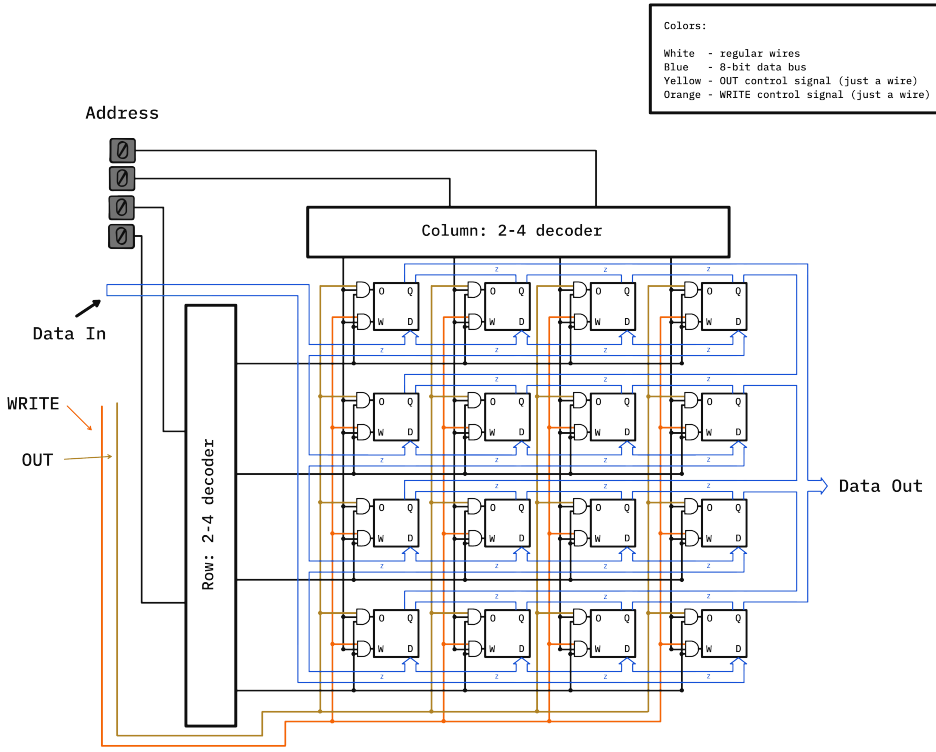


Diagram 8.5. A zoomed-out RAM diagram.

This is kind of a lot to unpack, so let me explain the high-level parts before we zoom in and take a closer look.

We have a blue `data in` bus that is fed into the bottom of all of the registers. We also have another blue `data out` bus that comes out of the top of all the registers and combines into one output. Every slot is connected to both buses, but only the selected slot is allowed to use them. If the slot's row and column are selected, it can either read from `data in` when `WRITE` turns on, or drive `data out` when `OUT` is on.

Let's look at one cell more closely:

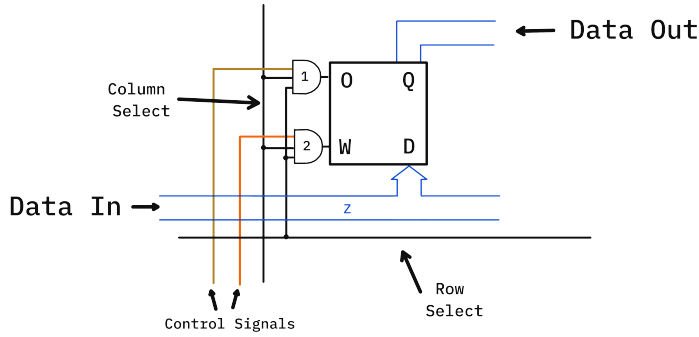


Diagram 8.6. A zoomed-in RAM cell diagram.

What AND gate 1 checks is, if `Row Select` and `Column Select`, and `OUT` is on, then that means we have selected that register to output its value, so we turn on that register's own `OUT`, and it will output something on the `data out` bus.

AND gate 2 checks, if `Row Select` and `Column Select`, and `WRITE` is on, then that means we have selected that register to write to, so we turn on that register's own `WRITE`, and it will write the data on the `data in` bus.

Both AND gates take in 3 inputs. If you are wondering how that works, think of two AND gates chained together.

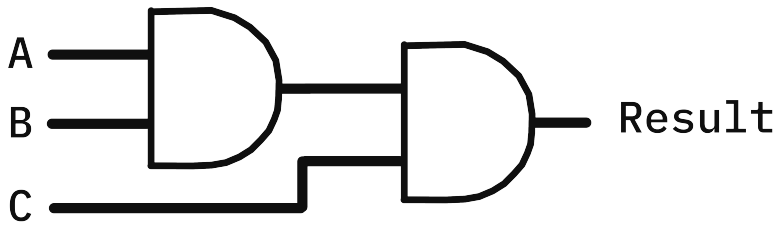


Diagram 8.7. A three-input AND gate.

So now that we have built RAM, let's pretend that instead of 16 registers, we have a RAM array with 256 registers. The same logic can be copied, just with two `4-16` decoders instead of two `2-4` decoders and 8 address inputs rather than 4.

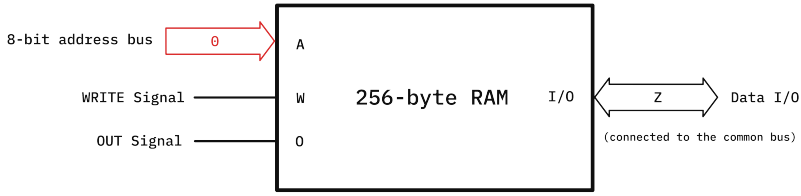


Diagram 8.8. Our RAM chip.

Technically, there are still two buses inside the RAM: a data-in path and a data-out path. That is basically what we saw with the register in the bus section (Diagram 7.5).

But drawing two separate data buses every time is cumbersome. From the outside, we can abstract this as one shared data bus with a double-headed arrow called `I/O`, which stands for input/output. It is practically just like having two buses, one for input, one for output.

When `WRITE` turns on, RAM copies the value from the data bus into the selected address. If `OUT` is on, RAM drives the selected address's value onto the data bus.

NOTE

From now on, instead of drawing registers connected to a common bus like this (Diagram 7.5), where we have a separate `D` and `Q`, I'll draw them with one double-headed `I/O` bus, like this:

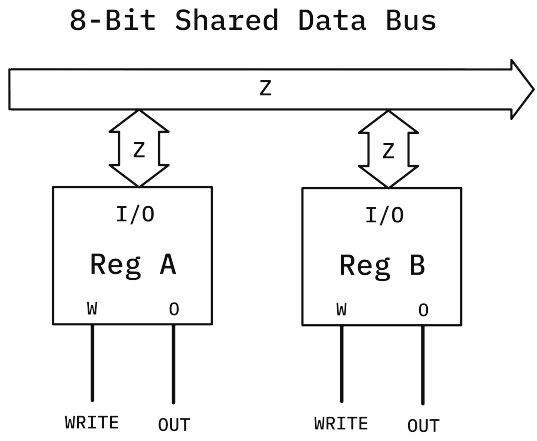


Diagram 8.9. I/O Registers.

They both are the same technically, just this is easier to draw.

Now back to the RAM chip. If you pay close attention to Diagram 8.8, you will notice that the address input is not directly connected to the common data bus.

That is intentional. The data bus is for moving values around the CPU. During a RAM operation, it needs to carry the value being written to RAM or the value being read from RAM. So it cannot also keep holding the address at the same time.

We need something to hold the address while the data bus is being used for the actual I/O. This is called the Memory Address Register, or MAR. The MAR is just a regular 8-bit register with no `OUT` control signal, as it is always outputting directly into RAM.

We first put an address on the data bus and turn on `MAR_WRITE`. The MAR stores that address. Then the MAR keeps sending that address to RAM, leaving the data bus free to carry the value being read or written.

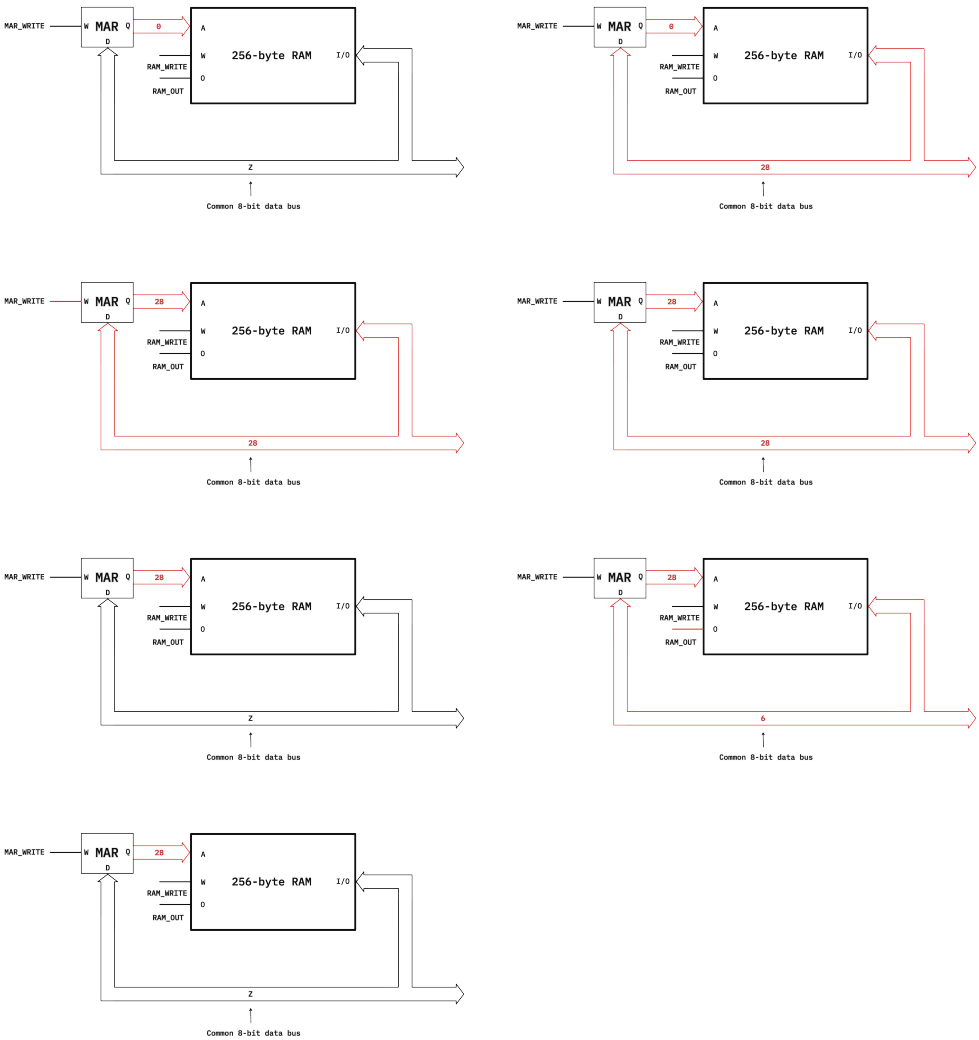


Diagram 8.10. How the MAR works.

So first, we put 28 onto the common bus. Enable `MAR_WRITE` and store that into the MAR. We then remove 28 from the common bus and enable `RAM_OUT`, and we get 6 as the value stored in slot 28. Cool.

9. The ALU

If we have a handful of registers and RAM, we can now move bytes around using this common data bus. But what we really need is a component that can "process" numbers, a component that can do arithmetic and logic. Thus we have "The Arithmetic and Logic Unit," or ALU for short. Imagine a chip where we could input two numbers, an operation, and output a result, along with some other information.

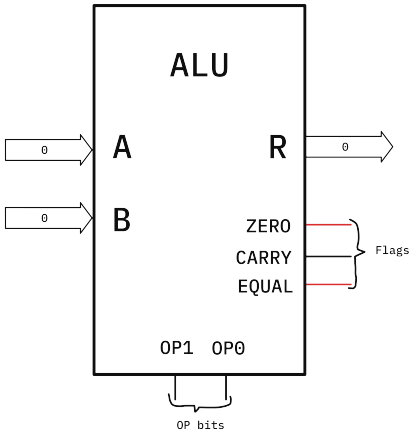


Diagram 9.1. The ALU chip.

For this CPU, I am keeping the ALU simple. It will have two operation-select bits, which gives us four possible operations:

OP	Operation	Meaning
00	ADD	output A + B
01	AND	output A AND B
10	OR	output A OR B
11	XOR	output A XOR B

The ALU will also output a few flags, which, if you remember, are just extra yes/no facts about the result or the inputs:

Flag	Turns on when
`ZERO`	the result is `00000000`
`CARRY`	addition spills past 8 bits
`EQUAL`	`A` and `B` are the same

So, in Diagram 9.1, we did `0 + 0`, which is 0, so the `ZERO` flag is on, and the `EQUAL` flag too because both inputs are equal.

Before we look at the internals of how the ALU works, we need to cover two things.

First, the ALU uses a new component called an 8-bit buffer. It is just a tri-state buffer for a whole bus: 8 of them side by side, all sharing one `OUT`. We use these to only let through the result we actually want on the output.

Also, you'll see me feeding two buses into a single logic gate. "How does that work?" you might think. Well, there are really just eight gates, one per bit. One gate takes `A0` and `B0`, the next takes `A1` and `B1`, and so on. Eight output wires, which is just another bus.

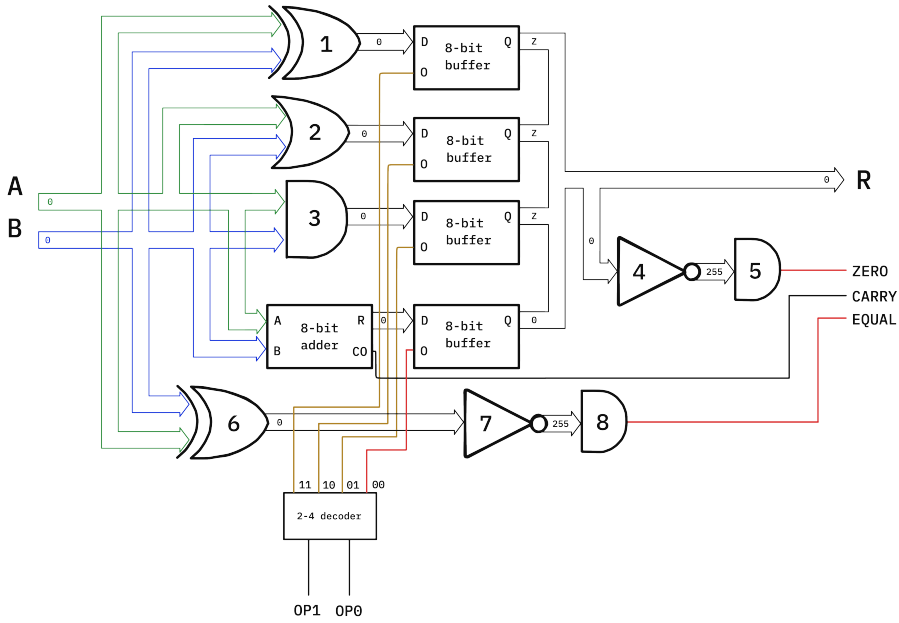


Diagram 9.2. ALU internals.

At its crux, the ALU works by routing `A` and `B` into all 4 operations at once, in this case XOR, OR, AND, and ADD. Each result goes into its own 8-bit buffer.

To decide which one to output, the `OP` bits go into a `2-4` decoder that enables exactly one of the 8-bit buffers onto the `R` bus.

Now for the flags. First, the `ZERO` circuitry. It consists of:

NOT → AND

The NOT gate on the right, labeled 4, is just like before: there are actually eight NOT gates, each flipping one wire of the bus. So it takes in a bus, and outputs a bus.

But the AND gate next to it, labeled 5, takes in a bus and outputs just one wire. Like the three-input AND from the RAM section (Diagram 8.7),

this AND gate takes in 8 inputs and produces 1 output. It just checks if all of its inputs are on.

So if we flip each bit, and then check if all of them are on, we get the `ZERO` flag. This makes sense because all the bits going into the AND gate can only be on if they were originally `00000000`, a.k.a. zero.

Next, let's cover the `EQUAL` circuitry. It consists of:

XOR → NOT → AND

The XOR gate, labeled 6, works by XORing each pair of bits from `A` and `B`, and producing one output bus. Remember, XOR outputs `0` when its two inputs are the same. So if `A` and `B` are equal, we get `00000000` as the output.

Then we flip the bits with gate 7, getting `11111111`, and if we AND them all together with gate 8, we can check if they are all true. If even one pair of bits differs, that wire ends up `0` after the flip, and the AND outputs `0`. Simple.

The `CARRY` flag is simple: we just connect the adder's carry-out, `CO`, directly to the flag output. Of course, it only means anything when we are actually adding. The ALU is complete. We can now piece together this whole CPU.

> **TRY IT**
> The ALU adds `11111111` and `00000001`. Which flags turn on?
> **Answer:** `ZERO` and `CARRY`. `EQUAL` stays off, because the two inputs are different.

10. The Big Picture

Here is a big-picture diagram of the whole computer.

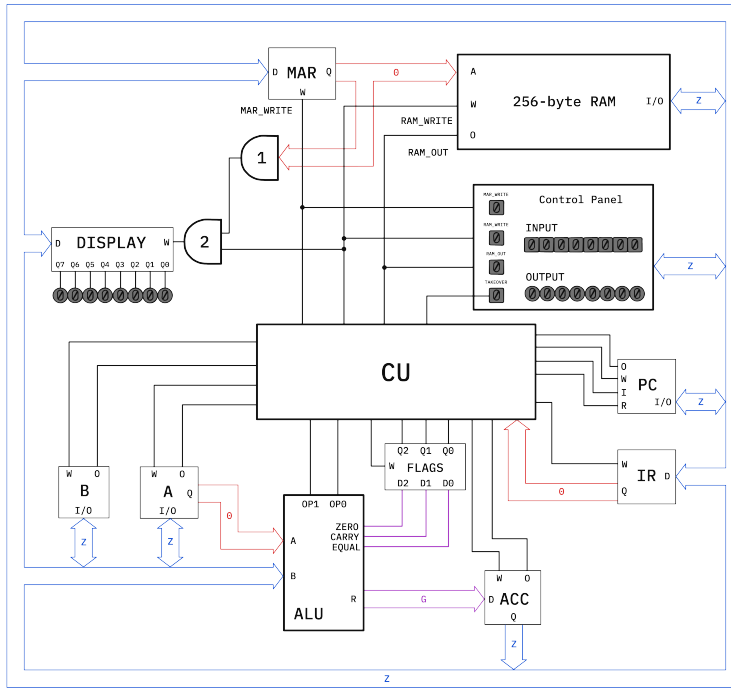


Diagram 10.1. The whole computer.

Hey? Doesn't this diagram look familiar? Well, yes, we have come full circle from the intro, except this time the CPU isn't something completely foreign. Of course we still have a lot to learn and build, but wow. Just wow.

Now comes the time where I finally explain to you curious readers what a CPU actually is. CPU is short for Central Processing Unit, and its job is to run programs: grab instructions from RAM one at a time, figure out what they mean, and carry them out. The CU is part of the CPU, and so are the ALU, the registers and the buses. RAM is not part of the CPU; the CPU just talks to it. Same for the DISPLAY and the Control Panel. In a

real computer, all of those core parts are usually packaged onto their own chip, which talks to RAM and everything else through separate wires. But as you can tell, I have included everything in this one diagram, from the DISPLAY and the Control Panel to the CU, so really it's a whole little computer.

Some quick notes on the diagram before we dig in.

> **NOTE**
> The 8-bit data buses are blue when not being driven. If a control wire is red, it means it's on. If a bus is red, it means it's being driven like normal.

But we have something new here that we haven't seen before.

Purple wires and buses mean garbage. This is not the same as `Z`. A `Z` wire is one that nothing is driving. A `G` wire is being driven, with an actual value on it. That value is just nonsense, hence the name garbage. So where does that garbage value come from in this diagram?

Remember, the ALU never stops computing. It always takes its inputs and has a result instantly. One input comes from register A (we will talk more about how register A works later), but the other input comes from the bus, and when nothing is driving the bus, the input is sitting at `Z`. The thing is, in this CPU an input at `Z` would just behave like 0. Our gates are built from relays, and a relay coil with nothing driving it is simply off, exactly as if you fed it 0. So really, with `OP` set to ADD like it is here, the ALU would be computing `A + 0` if the second input was `Z`.

But we mark it as garbage anyway, because in a real CPU made from transistors instead of relays, a `Z` value does not settle to a clean 0; it drifts and can get really funky. That is why we treat those values as garbage: we just ignore them. If the bus is being driven then we can of course use those values.

Now back to the diagram. Almost every single chip here, we have already built. The RAM and MAR combination, we have seen how that works previously (Diagram 8.10). A, B, ACC, IR, FLAGS, and DISPLAY are all just slight variations of registers. PC is an accumulator-style circuit, and we know how the ALU in the bottom left works (Diagram 9.2).

Let's examine each element of the CPU closely and see what details changed and why. First, register B. It is a normal edge-triggered 8-bit register with `WRITE` and `OUT` control wires. We will talk about its use in more detail later.

We then have register A, which is just like register B, but it also has a `Q` output in addition to `I/O`. This `Q` output is simply the value stored inside register A, bypassing the tri-state buffers and feeding directly into the first input of the ALU. So even though the `OUT` control signal is off, that `Q` is still driving the ALU's first input, in this case to 0.

Next we have the ALU. Its second input comes from the common bus, so the other number it works with is just whatever is on the bus at the time. Other than that it is as normal, but the flags are all going into a register called FLAGS. This register is a 3-bit register: it only stores 3 bits. These 3 bits are just the values for each flag. This 3-bit register has no `OUT` control signal, thus no tri-state buffers, so it is always outputting, but it is edge-triggered.

Then we can see the result of the ALU operation is being fed into another regular 8-bit register called ACC, short for accumulator, but that's just its name. It's not an actual accumulator circuit. That's just what this register is traditionally called in CPUs. It is slightly different to register B because its input doesn't come from the common bus, but from the ALU, so it has a separate `D` and `Q` instead of just one `I/O`. It still has the regular `WRITE` and `OUT` control signals though and like always is edge-triggered.

Register IR, which stands for Instruction Register, is a simple register without an `OUT` control signal. It takes input from the bus and outputs it into the CU. It is used to store the instruction so the CU can decode it and figure out what it means. We will talk about what exactly the CU is and the jobs of these different parts a little later on.

Next we have PC, which stands for Program Counter. All it does is store the address of the next byte that we need to fetch. It is a combination of an adder and a register, something like what we have seen previously (Diagram 6.9). The register part of PC has its regular control signals `WRITE` and `OUT`, and the `STEP` button from before has been replaced with an `I` control signal, which stands for increment. We also have the `R` signal

which resets the register back to 0 with some more logic gates and wires. Nothing too fancy. In the control unit diagrams later on, these two show up as `PC_INC` and `PC_RESET`.

Our MAR and RAM are the same as before, but if all of the bits stored in MAR are `1`, meaning the address is 255, then that first AND gate in the diagram, labeled 1, turns on. We then AND that with `RAM_WRITE`: if we are writing to address 255, we also write into the DISPLAY register.

What ends up happening is that DISPLAY stores whatever is stored in RAM address 255, and displays that value on 8 bulbs for us to see. You can think of this as our simple output for any programs we might write.

Then we have the Control Panel. Its job is to load a program into RAM in the first place. It can read and write to any RAM address it wants.

How it works is, first you flip the `TAKEOVER` switch, which inside the CU basically freezes the computer and resets PC. So, while `TAKEOVER` is on and the `RAM_OUT` button is off, the value on the panel's input switches is put onto the common bus. This is done with some logic gates and tri-state buffers, nothing too crazy. If `RAM_OUT` were on, then RAM would try to drive the bus at the same time as the Control Panel. That's why we need to make sure it's off before we can safely drive the bus.

From there you can hit `MAR_WRITE`, which stores that as the address you want to work with in RAM. To write, you then flip the switches to the value you want and press `RAM_WRITE`. To read, you instead press `RAM_OUT`, which makes the Control Panel stop driving the bus (using tri-state buffers), because RAM will then drive the bus, and the output bulbs will turn on to that value.

Also, the panel's buttons only actually drive their control wires if `TAKEOVER` is on, again using some tri-state buffers and simple logic gates. This is to ensure that while the CPU is running like normal, the CU and the panel don't drive the wires at the same time. This works the other way too: when `TAKEOVER` is on, the CU makes sure not to drive those RAM control wires, again using tri-state buffers.

Now, about this mysterious box labeled CU. What is it exactly? It is the control unit. Think back to most of our previous demos. Turn `OUT` on,

then `WRITE`, flip this control signal and then flip that control signal. The CU does all of that flipping for us. It turns the control wires on and off in the right order, at the right times, to make the computer actually work.

But how exactly does the computer work in the first place? First we load our program into RAM: a list of instructions, each one just a number. The control unit then fetches the instruction from RAM at the address stored in PC, so if PC is 0, it fetches the instruction at address 0 and stores it in IR. How? It copies PC into the MAR and flips on `RAM_OUT`, exactly like we have done before (Diagram 8.10). Next it figures out what that instruction means, and executes a bunch of steps to actually do that instruction, then it increments or changes PC and repeats that whole cycle.

This is kind of a simplified version of what our CPU will do, but we will dive into it in the following sections. You can think of this cycle as fetch, decode, execute. Fetching gets us the instruction, decoding figures out what it means, and executing actually flips those control wires on and off to accomplish that instruction.

But before we can figure out how the CU works, we need to figure out how instructions work. What does an instruction even look like?

Part 3: Execution

Before we continue, here is what we have covered in Part 2:

- A flip-flop stores a bit, and 8 of them make a register.
- A bus lets registers share wires, and tri-state buffers make sure only one component drives it at a time.
- RAM is a big array of registers, chosen by an address.
- The ALU does math and logic on two bytes and sets flags.

11. Instructions Are Numbers

Instructions tell the CPU what to do, and a list of instructions is a program. There are game programs, calculator programs, email programs. Everything on your computer is a program. Just a long, long list of instructions telling the CPU to do simple things like add these two numbers, move this value here, etc. All programs are just a list of instructions stored in RAM, each instruction coded as a number.

So here is what number each instruction in our CPU translates to:

#	Binary	Mnemonic	Bytes	Action
0	`0000`	`NOP`	1	do nothing
1	`0001`	`LOAD A, n`	2	next byte → A
2	`0010`	`LOAD A, [addr]`	2	RAM slot → A
3	`0011`	`STORE A, [addr]`	2	A → RAM slot
4	`0100`	`MOV B, A`	1	copy A into B
5	`0101`	`MOV A, B`	1	copy B into A
6	`0110`	`JMP addr`	2	addr → PC
7	`0111`	`JE addr`	2	if `EQUAL` flag: addr → PC
8	`1000`	`ADD`	1	A = A + B
9	`1001`	`AND`	1	A = A AND B
10	`1010`	`OR`	1	A = A OR B
11	`1011`	`XOR`	1	A = A XOR B
12	`1100`	`JC addr`	2	if `CARRY` flag: addr → PC
13	`1101`	`JZ addr`	2	if `ZERO` flag: addr → PC
14	`1110`	`HALT`	1	stop the clock

Table 11.1. The full instruction set.

A few notes on reading this table. The mnemonic is just a short name for the instruction, so we humans don't have to remember numbers. `n` stands for a number, and `addr` stands for an address in RAM. Brackets mean the RAM slot at that address, so `[addr]` is whatever is stored there: `[5]` means whatever is stored at address 5, but just `5` means literally the number 5. Without brackets, like in the jump instructions, `addr` is just the address number itself. The arrow → means "goes into". Bytes is how many bytes of RAM the instruction takes up. And the binary column only shows 4 bits because 4 bits are enough to number all 15 instructions. In RAM, the opcode still takes up a full byte. The top 4 bits are just `0000`.

The number for each instruction is called its opcode. Let's write a tiny program. We store `00000001` in address 0 of RAM. Then store `00000101` in address 1 of RAM. Then store `00000011` in address 2, then `11111111` in address 3, and finally `00001110` in address 4.

Our RAM looks like this:

```
0: 00000001
1: 00000101
2: 00000011
3: 11111111
4: 00001110
```

If PC, the program counter, starts at 0 (which it does after being reset by `TAKEOVER`), then what do you think this program would do? The first byte is `00000001`, which is 1, and the table says 1 is `LOAD A, n`: load the next byte into register A. The next byte is `00000101`, binary for 5, so those two bytes make up the instruction:

```
LOAD A, 5
```

The first byte tells us what the instruction is; the second tells us what value to load. The next instruction is 3, `STORE A, [addr]`, which basically puts the contents of `A` into the address of RAM that we specify. Again, the first byte tells us the instruction and the second byte tells us the address, in this case `11111111`, which is address 255.

Hey? Isn't address 255 different from the rest? Yep, if we write to address 255 we also write to the DISPLAY register where we can see that value on some light bulbs. Have a look at DISPLAY and the AND gates labeled 1 and 2 in Diagram 10.1 for a recap.

The last byte is 14, which is `HALT`. It stops the clock, the whole machine freezes, and the program is over.

But why do we even need to stop the CPU? Well, if we don't, PC would just keep counting up and the CPU would keep fetching. It doesn't know our program ended at address 4. It would march straight into the rest of RAM and run whatever is sitting there as if it were instructions, leftovers from the last program, or whatever junk the memory happened to have in it. We need a way to stop.

This program puts 5 on the output and stops. Pretty simple. As you can tell, some instructions take two bytes, and some only need one. A 2-byte

instruction is one where the first byte alone is not enough. `LOAD A, n` needs to know what value to load, so the next byte in RAM is that value.

> **TRY IT**
> How many bytes do `LOAD A, 5`, `STORE A, [255]`, and `HALT` physically take up in RAM?
> **Answer:** 5. Two each for `LOAD` and `STORE`, and one for `HALT`. That's why our first program fills addresses 0 to 4.

One more thing: the order things are written in. For `MOV`, the destination comes first. `MOV B, A` copies `A` into `B`, not the other way around. It is really more like a "copy", since `A` keeps its value, but we just call it move. `STORE` is the odd one out: `LOAD A` means the value is going into `A`, and `STORE A` means `A` is going out into memory.

> **TRY IT**
> What ends up on DISPLAY after `LOAD A, 3`, `MOV B, A`, `ADD`, `STORE A, [255]`, `HALT`?
> **Answer:** 6.

Now, about the jump instructions: `JMP`, `JE`, `JC`, and `JZ`. All of these instructions can set PC to the address stored in the second byte.

`JE`, `JC`, and `JZ` each check one flag. The flags from the last ALU instruction sit in the FLAGS register until the next ALU instruction overwrites them. So to make a decision, you run an ALU instruction, then jump based on what it found. If the flag is on, the jump sets PC to that address. If not, the program just carries on to the next instruction. This is the core element used in creating programs that can branch (if else statements) and loop.

For example:

```
0: LOAD A, 1
2: STORE A, [255]
4: MOV B, A
5: ADD
6: JMP 2
```

One note: the number before each instruction is just the address in RAM where it would be stored. That is why you can see it jump in twos for 2-byte instructions.

This jumping back to the start of the loop is done with a jump instruction, so whenever PC reaches 6, it jumps back to 2 and repeats all over again. Each loop, this program displays `A`, copies `A` into `B`, and then adds. Also, in case you missed it in the table, the ALU instructions always work on registers A and B: A is fed straight into the ALU, and B is put onto the bus by the CU. `ADD` adds them and puts the result back into A. Both ALU inputs are the same number, so `A` doubles every loop, and the output shows 1, 2, 4, 8, 16, all the way up to 128. On the 8 output bulbs, that looks like a single lit bulb going across the display. Pretty neat.

But then 128 + 128 overflows (like the third example in Diagram 5.12): `A` wraps around to 0. And from that point on, the program is doubling 0 forever, so the display becomes blank forever. The CPU gets stuck in a weird spot.

To solve this overflow problem, and in general to make decisions, we can use the flags to jump to different parts of our program depending on what the last ALU instruction found.

```
0: LOAD A, 1
2: STORE A, [255]
4: MOV B, A
5: ADD
6: JC 0
8: JMP 2
```

Now, when the addition overflows, the `CARRY` flag turns on, and `JC` catches it: we jump back to the start of the program, address 0, which loads 1, and the program starts all over again. Forever. On every normal loop, `JC` checks the flag, finds it off, and the program just carries on to the `JMP`.

Because this computer is built from relays, and thus pretty slow, we can probably watch the light walk across the display with our own eyes. If

we want to slow it down some, we can pad the program with some NOPs that waste clock cycles.

Real computers are insanely fast and usually have specialized timer hardware and such, but sometimes run "do-nothing loops" that just waste clock cycles for a set time.

Here is an example of a simple do-nothing delay loop:

```
0: LOAD A, 1
2: MOV B, A
3: LOAD A, 0
5: ADD
6: JC 10
8: JMP 5
10: (rest of the program)
```

All this loop does is add 1 to A over and over, 256 times, until the addition overflows and the CARRY flag lets it escape. It computes nothing useful. It just eats up time.

But we don't really need these for our slow computer. Just know real computers use a combination of timer hardware and loops like these to wait for the right amount of time.

These jump instructions can help us branch on certain conditions, letting our program decide stuff, and also run in loops.

But how would the CPU actually execute an instruction? What control wires would it flip, and in what order? Let's take the `MOV B, A` instruction as an example.

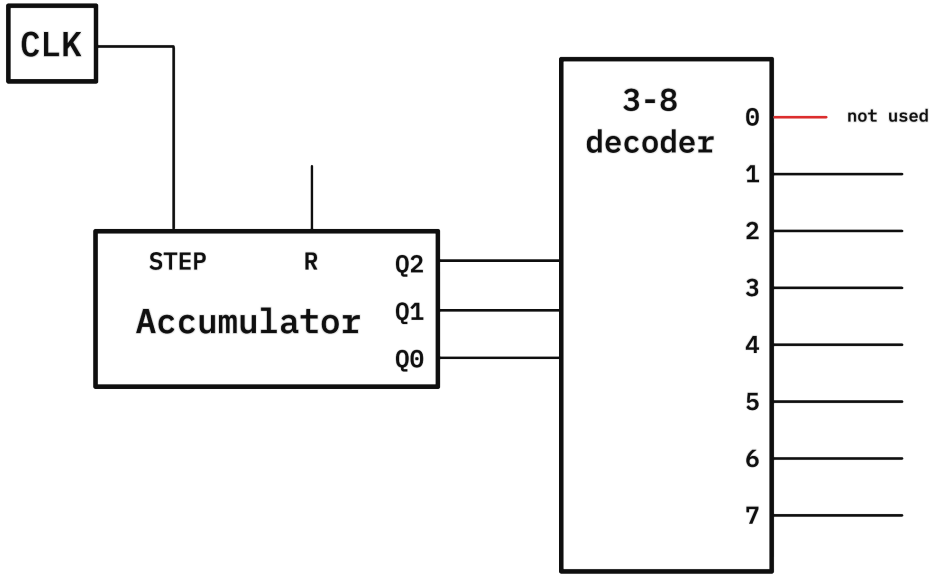
First the CU would flip the OUT signal on register A, driving the bus to whatever is stored in A. Then the CU would flip the WRITE signal on register B, writing the value on the bus into the register. Then it would turn off the OUT signal on register A, and boom, we are done. The CU had to do a few tiny things in exactly the right order to execute that instruction. Each of those tiny things is called a micro-step, and we will talk about them in more detail later on.

But, how does the CU know which wires to flip and when? That is what the next section is all about.

12. The Not-So-Clever Control Unit

The first thing the control unit needs is a step counter. Something that counts, step 0, then step 1, then step 2, and so on, back to step 0. That is the core flow of the control unit.

We can combine a tiny 3-bit accumulator and a `3-8` decoder. This will give us 8 steps, and because the accumulator overflows and naturally wraps back around to 0, the steps will loop correctly.



The full animation is at bytecolony.computer/writing/the-cpu/#f=cu-counter:1.

Diagram 12.1. The step counter.

The accumulator part of the step counter is connected to the clock (Diagram 6.10). The clock is the heartbeat of the computer, and it lives right here in the control unit.

Output 0 of the decoder is wired to nothing, so step 0 is just idle: the counter goes 1, 2, ... 7, 0, and every instruction takes 8 clock cycles whether it uses them or not.

One thing: remember the `TAKEOVER` switch (on the Control Panel in Diagram 10.1)? Well, when it is on, it basically resets this step counter's accumulator to 0 (that is why we don't actually do anything on step 0), so after a reset the first thing that happens is step 1. This also means `TAKEOVER` is a reset switch. Flick it on and off, and both PC and the step counter get reset to 0.

Each of the decoder's output wires turning on is 1 micro-step. Take `MOV B, A` from before: enable the output of `A` onto the bus, then enable `WRITE` on `B` to store that value.

If we tried to just turn `OUT` and `WRITE` on at the same time, the register might capture the value on the bus before the other register has started driving the bus. The bus hasn't had time to settle. So this won't work. What we want is to turn `OUT` on, wait for the bus to settle, and then turn `WRITE` on, right in the middle of `OUT` being on.

We could try spreading this across two steps: turn `OUT` on for step 1, then keep `OUT` on for step 2 and turn on `WRITE` too. The problem is that when the counter switches from one step to the next, the decoder briefly flickers through the wrong output, so `OUT` can drop at the exact moment `WRITE` fires. And every move would now cost two steps instead of one. It's not a good way to do it.

The best approach to this is just taking our clock signal and flipping it:

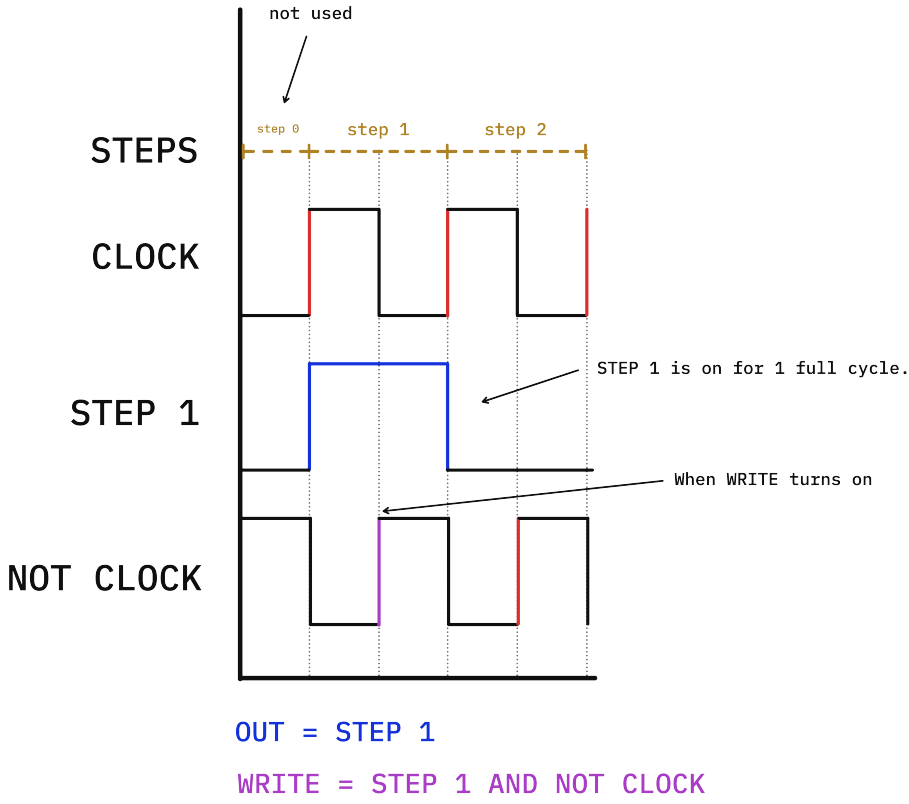


Diagram 12.2. Micro-step timing diagram.

Purple marks when `WRITE` turns on, and blue is when `OUT` is on. Each step changes on a clock edge, and each step is a signal, one full clock cycle long, until the next rising edge.

`OUT` is directly connected to that step line. If we enable `OUT` on step 1, the bus is driven for that full cycle.

`WRITE` is ANDed with `NOT CLOCK`, like `step 1 AND NOT CLOCK`. The thing is, step and `NOT CLOCK` don't come on at the same time. The step

line is already on, and then halfway through the cycle `NOT CLOCK` goes up, and that is when the AND gate's output goes up. That is the edge `WRITE` needs, right in the middle, after `OUT` has had some time to settle.

Now that we have covered that, let's figure out what should happen on each step. The first thing the CU will have to do for every instruction, no matter what, is to actually fetch that instruction from RAM and store it in IR, the instruction register.

Remember, PC holds the address of the next byte that we need to fetch. Then after we have fetched that byte, it usually gets incremented by 1, but with jump instructions, the value in PC gets changed, so the program starts executing from somewhere else.

To fetch the next instruction, first we would need to put the value in PC into the MAR. Then activate `RAM_OUT`, putting that instruction on the common bus, and store that value inside the IR register. Then we increment PC so it points at the next byte in RAM. Sometimes this next byte is a new instruction and sometimes it's the second part of a 2-byte instruction.

So to accomplish this, we only need two simple micro-steps:

1. `PC_OUT`, `MAR_WRITE`
2. `RAM_OUT`, `IR_WRITE`, `PC_INC`

We know how we can output and write in one micro-step, but three things? How can we increment PC too? Incrementing PC has nothing to do with the bus and doesn't disrupt anything else going on, so we just can. And notice, these two micro-steps are exactly the same for every single instruction.

One thing I am not drawing in the following diagrams: every `WRITE` signal is ANDed with `NOT CLOCK`, like we just discussed. The same goes for PC's `I` signal: incrementing stores a value too. It just comes from PC's own little adder instead of the bus, so it waits for `NOT CLOCK` like every other store.

Drawing that same little AND gate nine times would clutter everything up, so instead:

> NOTE
> Every pin colored pink has that AND gate secretly. A pink pin only actually turns on at `NOT CLOCK`.

One more thing: I have used tri-state buffers before the RAM and MAR control signals, and I have kept the enable on for now. This is related to interfering with the Control Panel and `TAKEOVER`. You can mostly ignore this for now, but it will come back later.

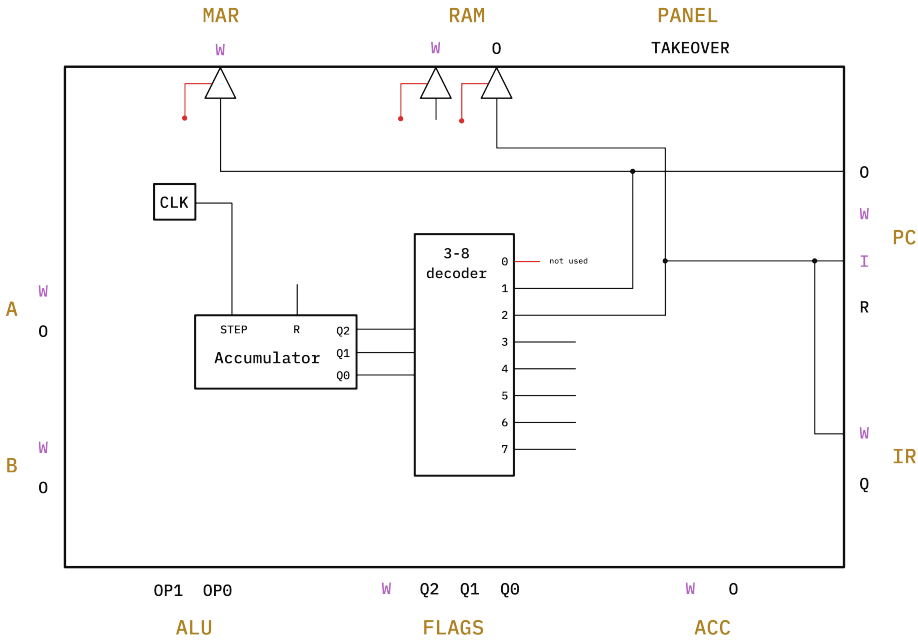


Diagram 12.3. The first two micro-steps.

That is the fetch.

13. Decode & Execute

Steps 1 and 2 are the fetch, and they are the same for every instruction. The rest of the micro-steps depend on the instruction we want to execute, so we use another decoder, this time a `4-16` decoder. The opcode is stored in a full byte, 8 bits, but it only needs 4 bits, so only the bottom 4 bits of IR are wired into the decoder and the top 4 are ignored. We only have 15 instructions (0–14, Table 11.1), so we can ignore output 15 of the decoder, and also output 0, since `0` is a `NOP`, which, if we remember, does nothing.

Let's first focus on the 2-byte instructions, the instructions that need to fetch a second byte. Instructions like `LOAD A, n`, `LOAD A, [addr]`, `STORE A, [addr]`, `JMP addr`, `JE addr`, `JC addr`, `JZ addr`.

Because all of these instructions need to fetch another byte, the third micro-step is always the same:

3. `PC_OUT`, `MAR_WRITE`

We can check, "is this instruction one of the 2-byte instructions?", using a giant OR gate, and if it is, on step 3, we can just execute the same micro-step. Steps 4 and 5 of these instructions all vary slightly, but nothing a few logic gates can't solve.

#	Instruction	Step 4	Step 5
1	`LOAD A, n`	`RAM_OUT`, `A_WRITE`, `PC_INC`	Nothing
2	`LOAD A, [addr]`	`RAM_OUT`, `MAR_WRITE`, `PC_INC`	`RAM_OUT`, `A_WRITE`
3	`STORE A, [addr]`	`RAM_OUT`, `MAR_WRITE`, `PC_INC`	`A_OUT`, `RAM_WRITE`
6	`JMP addr`	`RAM_OUT`, `PC_WRITE`	Nothing
7	`JE addr`	`RAM_OUT`, `PC_WRITE` if `EQUAL` else `PC_INC`	Nothing
12	`JC addr`	`RAM_OUT`, `PC_WRITE` if `CARRY` else `PC_INC`	Nothing
13	`JZ addr`	`RAM_OUT`, `PC_WRITE` if `ZERO` else `PC_INC`	Nothing

And that "if else" in the jump rows? It is just a NOT gate and two more AND gates. If the flag is on, then we do `PC_WRITE`. If the flag is off, we do `PC_INC`. In simple terms: if we want to actually do that jump, we store the new address into PC. If we don't, we just increment PC and move on with our lives.

Feel free to try and step through this diagram, or don't, it's a giant mess:

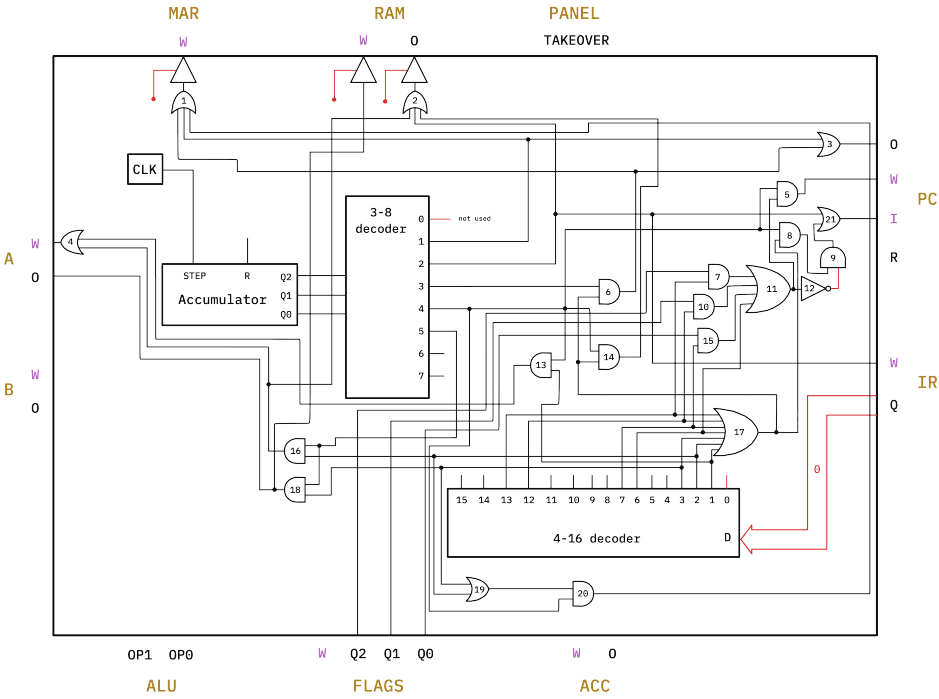


Diagram 13.1. All of the 2-byte instructions.

I know this is quite overwhelming, so let me break it down for you. We have steps 1 and 2 wired just like before to fetch our instruction. Gate 17, the OR gate, basically just checks if we have a 2-byte instruction or not, and some of our other gates use that answer. Here is what every gate in the diagram is checking for, and what it turns on:

Gate	Checks	Turns on
17	opcode is 1, 2, 3, 6, 7, 12 or 13	"2-byte"
6	step 3 and 2-byte	`PC_OUT`, `MAR_WRITE`
14	step 4 and 2-byte	`RAM_OUT`
13	step 4 and `LOAD A, n`	`A_WRITE`
19	`LOAD A, [addr]` or `STORE A, [addr]`	feeds gate 20
20	step 4 and gate 19	`MAR_WRITE`
7, 10, 15	`JE` and `EQUAL`, `JC` and `CARRY`, `JZ` and `ZERO`	feed gate 11
11	`JMP`, or any of 7, 10, 15	"jump"
5	step 4 and jump	`PC_WRITE`
12	NOT jump	feeds gate 9
8	step 4 and 2-byte	feeds gate 9
9	gate 8 and gate 12	`PC_INC`
16	step 5 and `LOAD A, [addr]`	`RAM_OUT`, `A_WRITE`
18	step 5 and `STORE A, [addr]`	`A_OUT`, `RAM_WRITE`
1, 2, 3, 4, 21	collectors, see below	`PC_OUT`, `MAR_WRITE`, `RAM_OUT`, `A_WRITE`, `PC_INC`

So on step 3, if we have a 2-byte instruction, then we enable `PC_OUT` and `MAR_WRITE`, and the `MAR_WRITE` of course fires on `NOT CLOCK`. Step 3 is the same for all 2-byte instructions since we always need to fetch that second byte, but what we do with that second byte changes.

On step 4, every 2-byte instruction needs to use that second value, so we always call `RAM_OUT` if it is a 2-byte instruction. Then each instruction does something slightly different with that number.

Instruction 1 just stores it straight into register A with `A_WRITE`, gated of course, only actually enabling on `NOT CLOCK`.

Instructions 2 and 3 are different. For those, that second value is not the number we want. It is the address of the number we want. So instead of storing it, we feed it into the MAR. Now the MAR is storing the address of the right slot, all ready for step 5 to do the read or write.

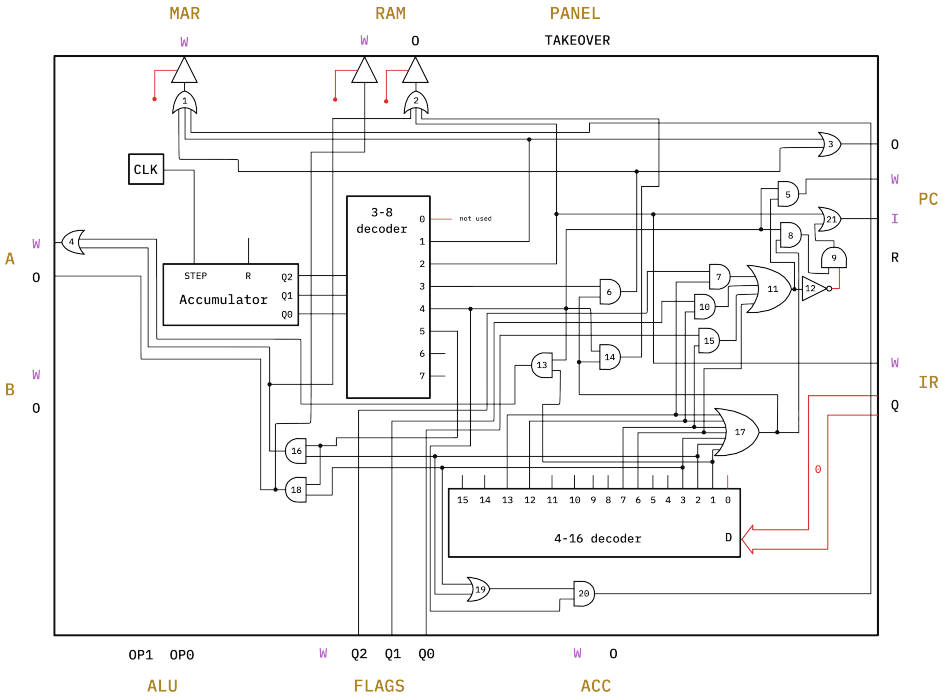
Now the jumps. For these, the second value of the instruction is a target address, so it needs to go into the PC. But first we have to figure out whether the jump actually happens. Gates 7, 10, and 15 do that: each one pairs a jump with its flag, so `JE` is checked against `EQUAL`, `JC` against `CARRY`, and `JZ` against `ZERO`. Those, plus a plain `JMP` which always jumps, all feed into gate 11, one big OR. If any of them fire, gate 11 turns on, meaning "we need to jump".

From there it is simple. If we need to jump, we enable `PC_WRITE` on step 4, loading the target address into PC, so then the program will start executing from that target address on the next fetch. If we don't need to jump, gate 12 flips that signal, and we enable `PC_INC` instead, letting the program carry on to the next instruction. That is the whole "if else" we were talking about in that table.

At this point only two instructions even use step 5: `LOAD A, [addr]` and `STORE A, [addr]`. Gates 16 and 18 check for those, then the appropriate control wires get turned on.

Also, you might notice, control signals that get activated for different reasons have a "collector" OR gate instead of everything being wired together. This is just to prevent a short-circuit, one wire driving low and another driving high. A simple OR gate solves this. Gates 1, 2, 3, 4, and 21 are these collectors.

Here is a little demo that shows the process of executing the instruction `LOAD A, [addr]`. We first have to fetch the instruction, then fetch the `[addr]`, then store whatever lives at that address into register A.



The full animation is at bytecolony.computer/writing/the-cpu/#f=cu-internals-two-demo:1.

Diagram 13.2. Executing `LOAD A, [addr]`.

When the label for a control signal turns red, that means it's on (pink ones still wait for `NOT CLOCK`). Next we do the `MOV` instructions.

#	Instruction	Step 3
4	`MOV B, A`	`A_OUT, B_WRITE`
5	`MOV A, B`	`B_OUT, A_WRITE`

Quite simple:

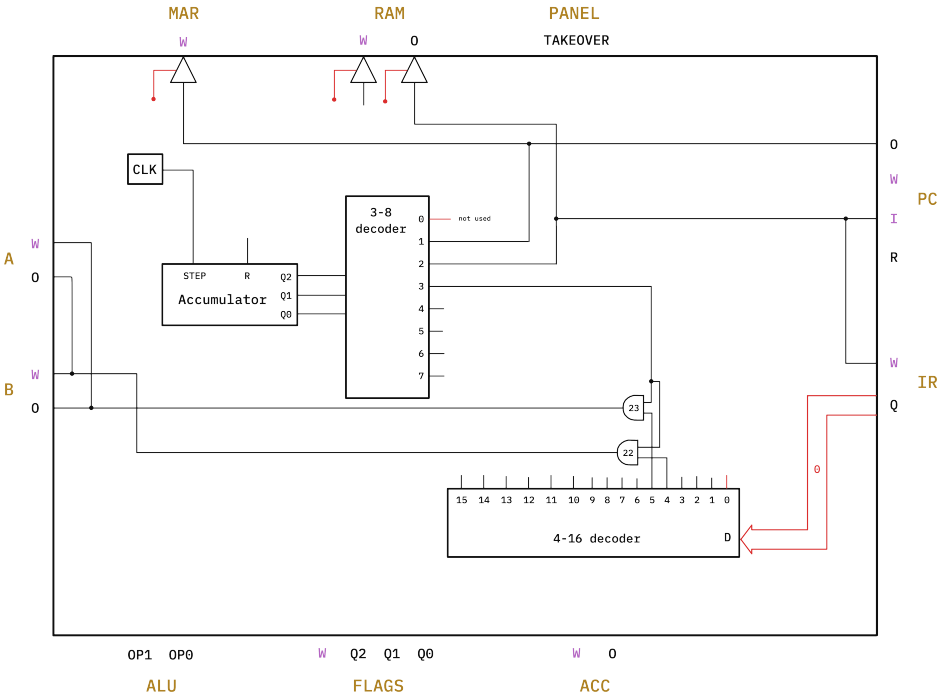


Diagram 13.3. The MOV instructions.

I have left out all of the 2-byte instructions from this diagram, just so it's easier to read. Next we need to decode all of our ALU instructions. Now this is where things get a bit weird, and I have some explaining to do.

In our CPU (Diagram 10.1), if you remember, register A has a special `Q` that bypasses the tri-state buffers, and feeds directly into the ALU. So to use our ALU, the programmer would first put their first number into register A using some instructions of their choice, and put their second number into register B. Then any ALU instruction would automatically use registers A and B as the two numbers, and the CU copies the result back into register A.

#	Instruction	Step 3	Step 4
8–11	ADD/AND/OR/XOR	B_OUT, ACC_WRITE, FLAGS_WRITE	ACC_OUT, A_WRITE

That is why on step 3 we only have B_OUT and no A_OUT since A is directly connected to the ALU already.

But what is ACC, you might be asking? ACC is kind of like a temporary register that the ALU uses to store its result. The ALU can't directly put the result onto the bus, since register B is driving the bus.

So on step 3, the CU lets B drive the bus, the ALU computes its result, and the CU stores the flags in FLAGS and the result in ACC. Then in step 4, when B stops driving the bus, we let ACC drive the bus and store that value into register A.

But now the big question: if all 4 of these instructions run the same micro-steps, what makes them different? Have a look at the last 2 bits of the number for each of these instructions in the instruction table (Table 11.1). Then have a look at the ALU's operation table.

Yep, the last 2 bits of each opcode directly match the ALU's OP bits for that specific operation. Must be a coincidence...

Anyway, because of this, we can just directly wire those 2 last bits into the ALU's OP bits. We only actually store the result of the ALU on these instructions. Sure, a STORE A, [addr] instruction might make the ALU XOR, but it doesn't matter because we don't store the result. It's garbage.

So here we are:

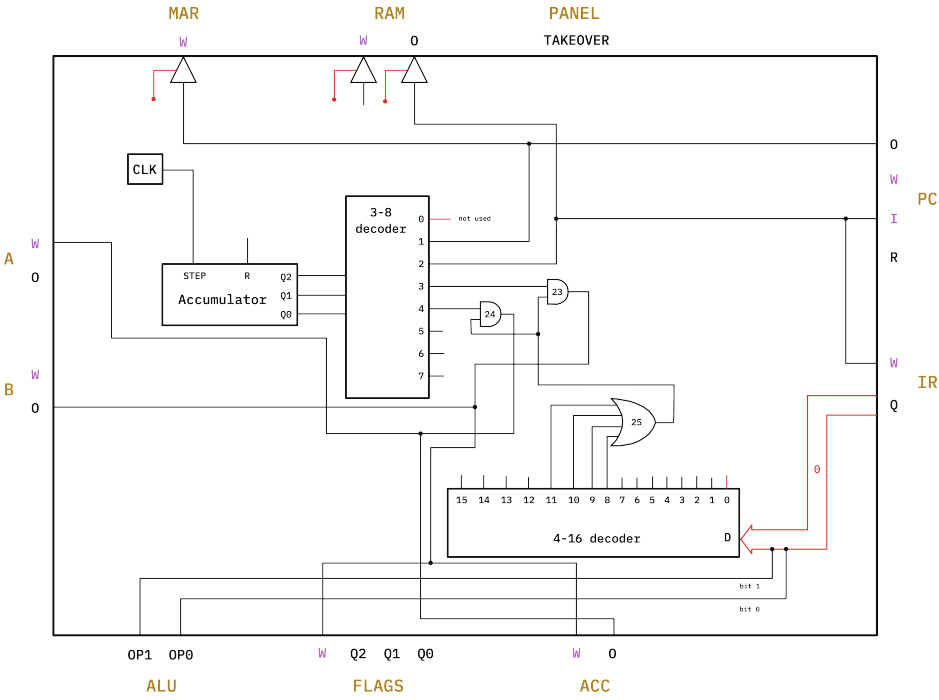


Diagram 13.4. The ALU instructions.

Again I am leaving out all the instructions apart from these 4, and again this wiring is a mess. All you need to know is that the bottom 2 bits of the opcode are tapped out and connected directly to the ALU's `OP` bits. And the rest of the logic just checks if it is an ALU instruction and if so, on the appropriate steps, it enables the correct control wires according to the table above.

One last thing: `HALT`. This instruction just stops the clock. The way we do this is by using an SR latch (Diagram 6.1), whose `Q` is inverted and then ANDed with the clock signal itself. So, on step 3 of instruction number 14, we just set that latch, and we have frozen the computer.

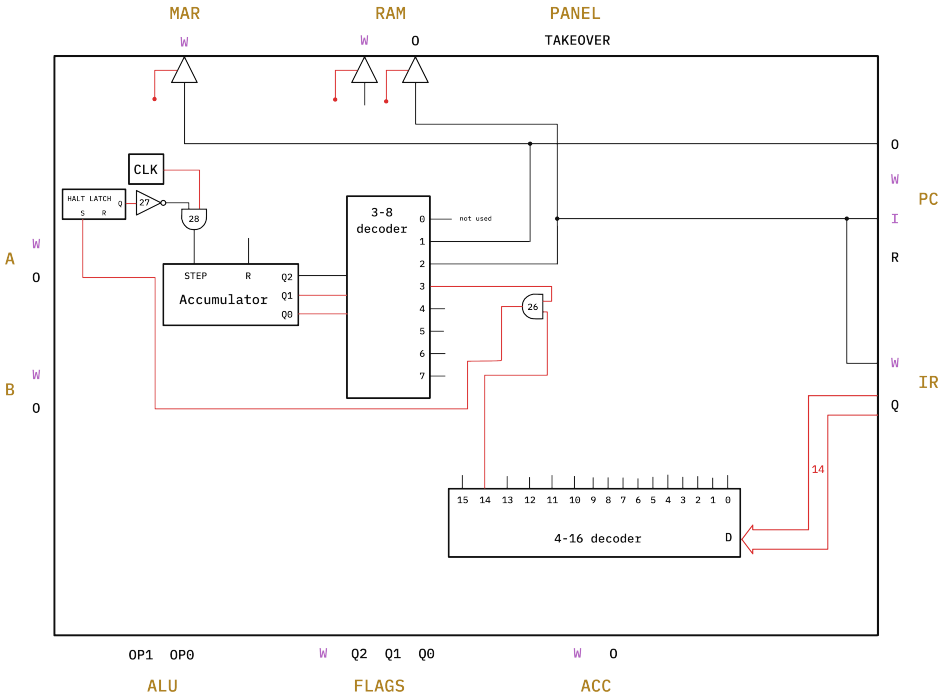


Diagram 13.5. The HALT instruction freezing the computer.

The clock keeps ticking, but the AND gate never lets it increment the counter, so the computer stays frozen. But how do we restart the computer? Well, remember the TAKEOVER switch I was telling you about? When we flick it on, it resets that latch. But its own inverted signal is ANDed in with the clock and the latch too, so a 3-input AND gate basically. So the computer stays frozen, but if you flick TAKEOVER off, then it will continue going.

TAKEOVER is also connected to PC_RESET, which resets the program counter to 0, so the computer will start executing from the first instruction when TAKEOVER is turned off. It also resets the step counter, moving it back to micro-step 0 in preparation for when the computer turns back on.

TAKEOVER also disables the CU's tri-state buffers and enables the panel's tri-state buffers for MAR_WRITE, RAM_WRITE, and RAM_OUT, so the panel can control those signals.

Here is how the HALT and TAKEOVER wiring would look in the CU:

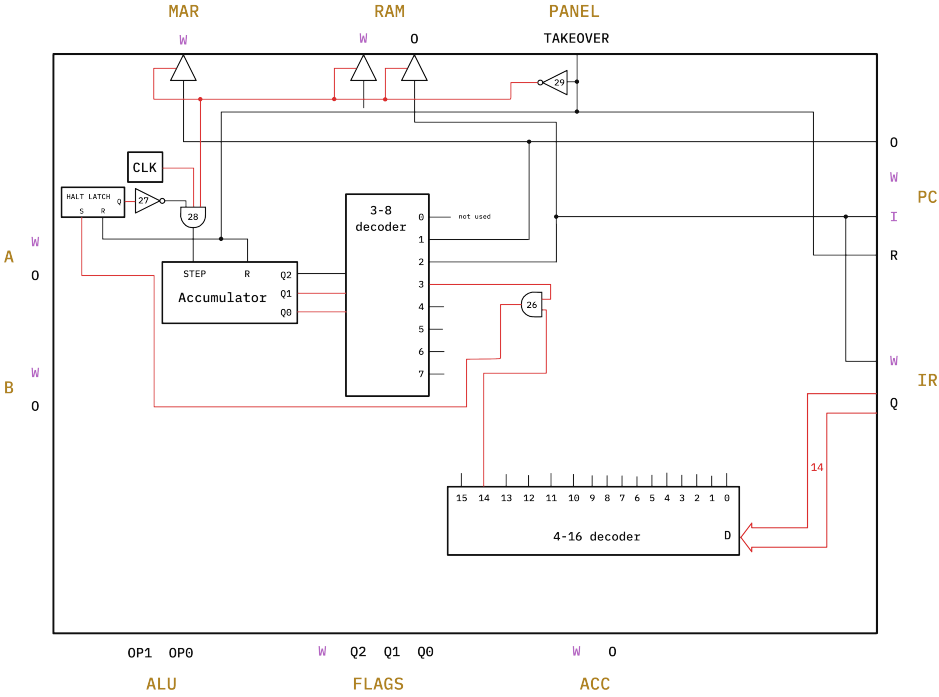


Diagram 13.6. The HALT and TAKEOVER wiring.

Wait, is that the whole CU? Umm, yes, that's it. Just combine all of those diagrams together, add in some extra OR gates and boom. That is the whole CU. You understand it, every wire of it. In the next short section, let's see how our computer can run a simple program.

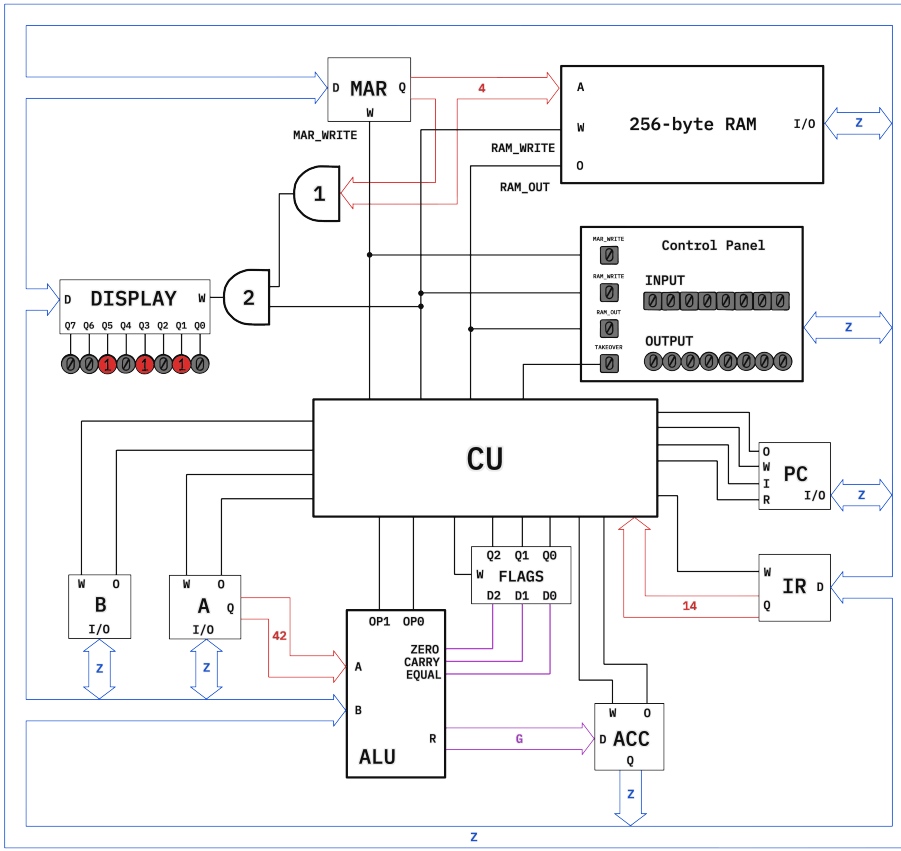
14. Finally, the Fun Part

That's it, huh? A whole computer. To test it, let's run a simple program on it.

Let's start with the simplest program imaginable, just putting a number on the DISPLAY bulbs:

```
0: LOAD A, 42
2: STORE A, [255]
4: HALT
```

Here is our CPU after it has finished executing it:



The full animation is at bytecolony.computer/writing/the-cpu/#f=program-demo:1.

Diagram 14.1. The CPU running a simple program.

And there it is, 42 sitting on the bulbs. Now let's try something real, a multiplication program. Put one number in slot 100, the other in slot 101, and the product ends up on the DISPLAY bulbs, as long as it fits in a byte.

```
0: LOAD A, 0
2: STORE A, [102]

4: LOAD A, 0
6: MOV B, A
7: LOAD A, [101]
9: ADD
10: JZ 30

12: LOAD A, [100]
14: MOV B, A
15: LOAD A, [102]
17: ADD
18: STORE A, [102]

20: LOAD A, 255
22: MOV B, A
23: LOAD A, [101]
25: ADD
26: STORE A, [101]

28: JMP 4

30: LOAD A, [102]
32: STORE A, [255]
34: HALT
```

The running total lives in slot 102. The program adds the value in slot 100 into slot 102, and then slot 101 counts down by one each time, until it hits 0.

This is basically all the program is:

```
total = 0 (slot 102)
while counter is not 0: (counter is slot 101)
    total = total + number (number is slot 100)
    counter = counter - 1
show total on the bulbs
halt
```

You might notice something weird:

```
20: LOAD A, 255
22: MOV B, A
23: LOAD A, [101]
25: ADD
26: STORE A, [101]
```

This little sequence first puts 255 into register B. Our CPU doesn't have an instruction to load directly into register B, so we have to load it into register A first and then move it over. Then it loads the value in slot 101 into register A, adds them, and stores the result back into slot 101. But why?

On this computer, adding 255 is just like subtracting 1! This is because the byte throws away anything past 255, so adding 255 and dropping the overflow is the same as taking 1 away. For example, let's say we have the number 5, we add 255, we get 260, but a byte runs out after 255 and rolls back over to 0, so 260 wraps around to 4.

One more oddity:

```
4: LOAD A, 0
6: MOV B, A
7: LOAD A, [101]
9: ADD
10: JZ 30
```

Our goal with this chunk of code is to just check if slot 101 is zero. If so, we are done and can jump to some other part of the program. Because this CPU is slightly more primitive than real CPUs, the programmer (me)

had to use some weird trickery to check if it is zero. In this case I just added 0 to the value. That doesn't change it, but the flags only update after an ALU instruction, and the `ZERO` flag only turns on if the answer is 0. The only way 0 + something gives 0 is if that something was 0, so now the `ZERO` flag tells us if slot 101 is 0.

My point with showing you all of these tricks is that, even though this CPU might not be the most advanced, it can do real things.

Modern CPUs have proper solutions to things like subtraction, negative numbers, division, more flags, and a lot more that smart computer science people just thought of. You can go learn about them yourself. But here is the thing: those dedicated circuits exist mostly for speed. Our funky little CPU can already do a lot more than you'd think, just slower, with tricks like the ones above.

Modern CPUs are also just bigger in general. Instead of 8 wires in a bus, 64. Instead of 256 bytes of RAM, billions. Instead of 2 registers you can program with, dozens. Instead of a few ticks per second, billions. But it is the same core loop, fetch, decode, execute, just bigger and faster.

Our program could also be seen as just a bunch of numbers:

```
1 0 3 102 1 0 4 2 101 8 13 30 2 100 4 2 102 8 3 102
1 255 4 2 101 8 3 101 6 4 2 102 3 255 14
```

And in RAM, it is actually just this:

```
0: 00000001 12: 00000010 24: 01100101
1: 00000000 13: 01100100 25: 00001000
2: 00000011 14: 00000100 26: 00000011
3: 01100110 15: 00000010 27: 01100101
4: 00000001 16: 01100110 28: 00000110
5: 00000000 17: 00001000 29: 00000100
6: 00000100 18: 00000011 30: 00000010
7: 00000010 19: 01100110 31: 01100110
8: 01100101 20: 00000001 32: 00000011
9: 00001000 21: 11111111 33: 11111111
10: 00001101 22: 00000100 34: 00001110
11: 00011110 23: 00000010
```

Some user would have to manually sit there at the Control Panel flipping these values into addresses of RAM one at a time. In the next closing section, we will touch a little bit on modern things like keyboards and monitors, all that other stuff that really turns a CPU into a computer.

15. The End

So what about keyboards, monitors? All the things that turn a CPU into a computer. Well, you have already seen the core idea. Imagine the DISPLAY register, just much bigger: thousands of memory slots instead of one, where each number sets the color of one tiny pixel on a screen. A keyboard is the same thing but in reverse, a slot you read instead of write, where the value is a code for the last key pressed.

Real computers are a lot fancier about it. They have dedicated I/O buses, whole controller chips that talk to the devices so the CPU doesn't have to, and devices that can even write into RAM directly on their own. They also have storage, like a disk or SSD, which lets information persist even when the power goes off.

Then there is the operating system. It is the first program that runs when you turn your computer on, and it does a lot. It loads programs from storage into RAM and runs them. It switches between programs so fast that it creates the illusion of many things happening at once. It handles the keyboard, the screen, and the disk. It makes the programmer's life much easier. This is of course the very basics. Operating systems do a lot of interesting and complex things, so if you are curious, check out *Operating Systems: Three Easy Pieces* in the Further Reading section.

So to recap, a program is just a list of numbers sitting in RAM. The control unit fetches those numbers one at a time, decodes them, and executes each one as a few micro-steps, flipping control wires that move bytes between registers. Every one of those components is made up of simple logic gates, and every gate is made up of relays, or more commonly these days, transistors.

I hope you now understand why computers aren't "just" 1s and 0s, but a very tall pile of simple.

Appendix: Quick Reference

The instruction set

#	Binary	Mnemonic	Bytes	Action
0	`0000`	`NOP`	1	do nothing
1	`0001`	`LOAD A, n`	2	next byte → A
2	`0010`	`LOAD A, [addr]`	2	RAM slot → A
3	`0011`	`STORE A, [addr]`	2	A → RAM slot
4	`0100`	`MOV B, A`	1	copy A into B
5	`0101`	`MOV A, B`	1	copy B into A
6	`0110`	`JMP addr`	2	addr → PC
7	`0111`	`JE addr`	2	if `EQUAL` flag: addr → PC
8	`1000`	`ADD`	1	A = A + B
9	`1001`	`AND`	1	A = A AND B
10	`1010`	`OR`	1	A = A OR B
11	`1011`	`XOR`	1	A = A XOR B
12	`1100`	`JC addr`	2	if `CARRY` flag: addr → PC
13	`1101`	`JZ addr`	2	if `ZERO` flag: addr → PC
14	`1110`	`HALT`	1	stop the clock

ALU operations and flags

OP	Operation	Meaning
`00`	ADD	output `A + B`
`01`	AND	output `A AND B`
`10`	OR	output `A OR B`
`11`	XOR	output `A XOR B`

Flag	Turns on when
`ZERO`	the result is `00000000`
`CARRY`	addition spills past 8 bits
`EQUAL`	`A` and `B` are the same

Micro-steps

Every instruction starts with the same fetch:

1. `PC_OUT`, `MAR_WRITE`
2. `RAM_OUT`, `IR_WRITE`, `PC_INC`

Two-byte instructions all share step 3: `PC_OUT`, `MAR_WRITE`. Then:

#	Instruction	Step 4	Step 5
1	`LOAD A, n`	`RAM_OUT`, `A_WRITE`, `PC_INC`	Nothing
2	`LOAD A, [addr]`	`RAM_OUT`, `MAR_WRITE`, `PC_INC`	`RAM_OUT`, `A_WRITE`
3	`STORE A, [addr]`	`RAM_OUT`, `MAR_WRITE`, `PC_INC`	`A_OUT`, `RAM_WRITE`
6	`JMP addr`	`RAM_OUT`, `PC_WRITE`	Nothing
7	`JE addr`	`RAM_OUT`, `PC_WRITE` if `EQUAL` else `PC_INC`	Nothing
12	`JC addr`	`RAM_OUT`, `PC_WRITE` if `CARRY` else `PC_INC`	Nothing
13	`JZ addr`	`RAM_OUT`, `PC_WRITE` if `ZERO` else `PC_INC`	Nothing

One-byte instructions finish in one or two steps:

#	Instruction	Step 3	Step 4
4	`MOV B, A`	`A_OUT`, `B_WRITE`	Nothing
5	`MOV A, B`	`B_OUT`, `A_WRITE`	Nothing
8–11	`ADD/AND/OR/XOR`	`B_OUT`, `ACC_WRITE`, `FLAGS_WRITE`	`ACC_OUT`, `A_WRITE`

Glossary

ACC. Short for accumulator. That is just its name, though: it is not an actual accumulator with its own adder. It is the register that stores the ALU's result. Its input comes from the ALU instead of the bus, so it has a separate `D` and `Q`. 67

Accumulator. A circuit that feeds its own output back into an adder, so each step adds to the running total. This is the concept behind PC. 37

Address. The number of a slot in RAM. RAM has 256 slots, so addresses go from 0 to 255. 51

ALU. Arithmetic Logic Unit. Takes two numbers and performs an operation on them, chosen by its `OP` bits. It also outputs the flags. 61

AND gate. Outputs 1 only when all of its inputs are 1. Usually drawn with two inputs, but it can take three or more, even a whole bus; an AND gate with more than two inputs can be made by chaining two-input AND gates together. 9

Binary. A number system that represents values using only two digits, 0 and 1. Each column is a power of 2: ones, twos, fours, eights, and so on. 20

Bit. A single 0 or 1. One wire, driven high or low. 18

Bus. A group of 8 wires carrying one value together. A shared, or common, bus lets many components transfer data over the same wires. 44

Byte. A group of 8 bits. It can hold 256 different values, so the numbers 0 to 255. 19

Clock. A signal that turns on and off over and over at a steady rate. Each tick moves the computer forward one step. 43

Control signal. A wire that tells a component what to do, like `W` (write) or `O` (output). The CU turns these on and off. Named after the part they control, like `MAR_WRITE` or `RAM_OUT`. 51

Control unit (CU). The part of the CPU that flips the control signals in the right order and at the right times to make the computer work. 68

CPU. Central Processing Unit. The registers, ALU, buses, and control unit together. Technically, RAM is not part of the CPU, since the CPU is only the processing unit, but the two are usually taught together. 1

D latch. A latch with a data input `D` and an enable input `E`. While enable is on, `Q` copies `D`. When enable turns off, `Q` holds its value. 37

Decoder. Takes a number in binary and turns on exactly one of its output wires. A `2-4` decoder has 2 inputs and 4 outputs; a `4-16` decoder has 4 inputs and 16 outputs. 52

Driven high / driven low. A wire connected to `+` (a 1) or to `-` (a 0). A real 0 is driven low, not just left disconnected. 13

Edge-triggered. Only stores a new value at the instant a signal turns from off to on. That instant is called the rising edge. 39

8-bit buffer. Eight tri-state buffers side by side sharing one `OUT` signal. Lets a whole byte onto a bus only when enabled. 62

Fetch, decode, execute. The cycle the CPU repeats forever. Fetch copies the next instruction from RAM into IR. Decode figures out which instruction it is. Execute flips the control signals that carry it out. 69

Flags. Extra yes/no facts about the ALU's result. Stored in the FLAGS register. 32

Flip-flop. Two D latches chained together to make an edge-triggered circuit that stores one bit. The building block of a register. 39

Full adder. Adds three bits: `A`, `B`, and a carry-in. Made from two half adders and an OR gate. 24

Garbage. A value that means nothing, such as a result computed from a bus nobody is driving. Shown in purple and always ignored. 66

Ground. The common return point of a circuit, usually connected to the negative side of the battery. 10

Half adder. Adds two bits and outputs a sum bit (XOR) and a carry-out bit (AND). It cannot take a carry-in. 24

Instruction. One command for the CPU, stored as a number in RAM. Some instructions take one byte; others take a second byte holding a value or an address. 69

I/O. Input/output. A single double-headed bus used both to put a value into a component and to read one out. 58

IR. Instruction Register. Holds the instruction currently being run so the CU can decode it. 67

Logic gate. A small circuit that answers a yes/no question about its inputs. The basic gates are AND, OR, NOT, NAND, NOR, and XOR. 10

MAR. Memory Address Register. Holds the address RAM should read from or write to, so the data bus stays free to carry the data itself. 59

Micro-step. One small action inside an instruction, lasting one clock cycle, like "copy PC into MAR". Every instruction is a short sequence of micro-steps. 76

NAND / NOR. AND and OR with their outputs flipped. 16

NOT gate. Flips its input: 1 becomes 0, and 0 becomes 1. 15

`NOT CLOCK`. The clock signal flipped. Writes are ANDed with it so values are stored halfway through a step, after the bus has settled. 79

Opcode. The number that says which instruction it is, such as `0001` for `LOAD A, n`. The bottom 4 bits of an instruction byte. 72

OR gate. Outputs 1 when any of its inputs is 1. 7

Overflow. When a result is too big to fit in one byte and the extra bit spills out as a carry. 32

PC. Program Counter. Holds the address of the next byte to fetch, and normally counts up by one after each fetch. Jump instructions overwrite it. 67

Program. A list of instructions sitting in RAM, run one after another. 69

RAM. Random Access Memory. Many registers in a grid, each with its own address. Any address can be read or written directly. 51

Register. Eight flip-flops side by side: a component that stores one byte and is edge-triggered. Usually has `W` (write) and `O` (output) control signals. 41

Relay. A switch flipped by electricity. Current through a coil makes a magnet that pulls a metal arm, closing a second circuit. 8

SR latch. The simplest circuit that stores one bit. `SET` makes `Q` 1, `RESET` makes `Q` 0, and with neither pressed it remembers. 35

Step counter. The counter inside the CU that tracks which micro-step the current instruction is on. 77

`TAKEOVER`. The control panel switch that freezes the CPU and resets PC. While it is on, the panel's buttons control `MAR_WRITE`, `RAM_WRITE`, and `RAM_OUT` instead of the CU, so a person can load a program by hand. 68

Tri-state buffer. A gate with three possible outputs: driven high, driven low, or disconnected (`Z`). Lets many components share one bus without short-circuiting. 45

XOR gate. Exclusive OR. Outputs 1 when exactly one input is 1. 24

`Z`. High impedance: a wire that nothing is driving. Not driven high or low, just disconnected. 45

Further Reading

Lessons in Electric Circuits, Volume I: DC by Tony R. Kuphaldt. A textbook on electricity. If the electricity basics at the start got you curious, this goes much deeper into voltage, current, resistance, and a lot more. Really loved this book, amazing read. Free online at allaboutcircuits.com.

Code: The Hidden Language of Computer Hardware and Software by Charles Petzold. The classic. This book is what got me excited about CPU architecture and was my introduction to the field. Goes much deeper, with some history too if that interests you. Highly recommended.

The Elements of Computing Systems (Nand2Tetris) by Noam Nisan and Shimon Schocken. A project course where you build a computer, an assembler, a compiler, and an operating system yourself, starting from NAND gates. Quite demanding; consider it if you want to go deep into computer science. Free materials at nand2tetris.org.

Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. The next step up the stack. This book breaks down how operating systems work in three easy pieces. Clear, funny, and highly recommended. Free online at ostep.org.

What Every Programmer Should Know About Memory by Ulrich Drepper. A great paper on how memory really works. It expands on the RAM ideas here, covering DRAM, caches, and a lot more. Free online.

Acknowledgments

My dad read the drafts and told me which parts were hard to understand or unclear. My family gave opinions and feedback on the cover mockups. Any mistakes that are left are mine.