



Registers and RAM: Crash Course Computer Science #6

Crash Course: Computer Science

<https://youtube.com/watch?v=fpnE6UAfbtU>

<https://nerdfighteria.info/v/fpnE6UAfbtU>

===== (00:00) to (02:00) =====

Hi, I'm Carrie Anne and welcome to Crash Course Computer Science. So last episode, using just logic gates, we built a simple ALU, which performs arithmetic and logic operations, hence the 'A' and the 'L'. But of course, there's not much point in calculating a result only to throw it away - it would be useful to store that value somehow, and maybe even run several operations in a row. That's where computer memory comes in!

If you've ever been in the middle of a long RPG campaign on your console, or slogging through a difficult level on Minesweeper on your desktop, and your beloved dog came by, tripped and pulled the power cord out of the wall, you know the agony of losing all your progress. Condolences. But the reason for your loss is that your console, your laptop and your computers make use of Random Access Memory, or RAM, which stores things like game state - as long as the power stays on. Another type of memory, called persistent memory, can survive without power, and it's used for different things; We'll talk about the persistence of memory in a later episode. Today, we're going to start small - literally by building a circuit that can store one single bit of information. After that, we'll scale up, and build our very own memory module, and we'll combine it with our ALU next time, when we finally build our very own CPU!

INTRO

All of the logic circuits we've discussed so far go in one direction - always flowing forward - like our 8-bit ripple adder from last episode. But we can also create circuits that loop back on themselves. Let's try taking an ordinary OR gate, and feed the output back into one of its inputs and see what happens. First, let's set both inputs to 0. So 0 OR 0 is 0, and so this circuit always outputs 0. If we were to flip input A to 1. 1 OR 0 is 1, so now the output of the OR gate is 1. A fraction of a second later, that loops back around into input B, so the OR gate sees that both of its inputs are now 1. 1 OR 1 is still 1, so there is no change in output. If we flip input A back to 0, the OR gate still outputs 1.

===== (02:00) to (04:00) =====

So now we've got a circuit that records a "1" for us. Except, we've got a teensy tiny problem - this change is permanent! No matter how hard we try, there's no way to get this circuit to flip back from a 1 to a 0. Now let's look at this same circuit, but with an AND gate instead. We'll start inputs A and B both at 1. 1 AND 1 outputs 1 forever. But, if we then flip input A to 0, because it's an AND gate, the output will go to 0. So this circuit records a 0, the opposite of our other circuit. Like before, no matter what input we apply to input A afterwards, the circuit will always output 0. Now we've got circuits that can record both 0s and 1s. The key to making this a useful piece of memory is to combine our two circuits into what is called the AND-OR Latch. It has two inputs, a "set" input, which sets the output to a 1, and a "reset" input, which resets the output to a 0. If set and reset are both 0, the circuit just outputs whatever was last put in it. In other words, it remembers a single bit of information! Memory! This is called a "latch" because it "latches onto" a particular value and stays that way. The action of putting data into memory is called writing, whereas getting the data out is called reading.

Ok, so we've got a way to store a single bit of information! Great! Unfortunately, having two different wires for input - set and reset - is a bit confusing. To make this a little easier to use, we really want a single wire to input data, that we can set to either 0 or 1 to store the value. Additionally, we are going to need a wire that enables the memory to be either available for writing or "locked" down - which is called the write enable line. By adding a few extra logic gates, we

can build this circuit, which is called a Gated Latch since the "gate" can be opened or closed. Now this circuit is starting to get a little complicated. We don't want to have to deal with all the individual logic gates... so as before, we're going to bump up a level of abstraction, and put our whole Gated Latch circuit in a box -- a box that stores one bit.

Let's test out our new component! Let's start everything at 0. If we toggle the Data wire from 0 to 1 or 1 to 0, nothing happens - the output stays at 0. That's because the write enable wire is off, which prevents any change to the memory.

===== (04:00) to (06:00) =====

So we need to "open" the "gate" by turning the write enable wire to 1. Now we can put a 1 on the data line to save the value to our latch. Notice how the output is now 1. Success! We can turn off the enable line and the output stays as 1. Once again, we can toggle the value on the data line all we want, but the output will stay the same. The value is saved in memory. Now let's turn the enable line on again use our data line to set the latch to 0. Done. Enable line off, and the output is 0. And it works!

Now, of course, computer memory that only stores one bit of information isn't very useful -- definitely not enough to run Frogger, or anything, really. But we're not limited to using only one latch. If we put 8 latches side-by-side, we can store 8 bits of information like an 8-bit number. A group of latches operating like this is called a register, which holds a single number, and the number of bits in a register is called its width. Early computers had 8-bit registers, then 16, 32, and today, many computers have registers that are 64-bits wide. To write to our register, we first have to enable all of the latches. We can do this with a single wire that connects to all of their enable inputs, which we set to 1. We then send our data in using the 8 data wires, and then set enable back to 0, and the 8 bit value is now saved in memory. Putting latches side-by-side works ok for a small-ish number of bits. A 64-bit register would need 64 wires running to the data pins, and 64 wires running to the outputs. Luckily we only need 1 wire to enable all the latches, but that's still 129 wires. For 256 bits, we end up with 513 wires!

The solution is a matrix! In this matrix, we don't arrange our latches in a row, we put them in a grid. For 256 bits, we need a 16 by 16 grid of latches with 16 rows and columns of wires. To activate any one latch, we must turn on the corresponding row AND column wire. Let's zoom in and see how this works.

===== (06:00) to (08:00) =====

We only want the latch at the intersection of the two active wires to be enabled, but all of the other latches should stay disabled. For this, we can use our trusty AND gate! The AND gate will output a 1 only if the row and the column wires are both 1. So we can use this signal to uniquely select a single latch. This row/column setup connects all our latches with a single, shared, write enable wire. In order for a latch to become write enabled, the row wire, the column wire, and the write enable wire must all be 1. That should only ever be true for one single latch at any given time. This means we can use a single, shared wire for data. Because only one latch will ever be write enabled, only one will ever save the data -- the rest of the latches will simply ignore values on the data wire because they are not write enabled. We can use the same trick with a read enable wire to read the data later, to get the data out of one specific latch. This means in total, for 256 bits of memory, we only need 35 wires - 1 data wire, 1 write enable wire, 1 read enable wire, and 16 rows and columns for the selection. That's significant wire savings!

But we need a way to uniquely specify each intersection. We can think of this like a city, where you might want to meet someone at



Registers and RAM: Crash Course Computer Science #6

Crash Course: Computer Science

<https://youtube.com/watch?v=fpnE6UAfbtU>

<https://nerdfighteria.info/v/fpnE6UAfbtU>

12th avenue and 8th street -- that's an address that defines an intersection. The latch we just saved our one bit into has an address of row 12 and column 8. Since there is a maximum of 16 rows, we store the row address in a 4 bit number. 12 is 1100 in binary. We can do the same for the column address: 8 is 1000 in binary. So the address for the particular latch we just used can be written as 11001000. To convert from an address into something that selects the right row or column, we need a special component called a multiplexer -- which is the computer component with a pretty cool name at least compared to the ALU. Multiplexers come in all different sizes, but because we have 16 rows, we need a 1 to 16 multiplexer. It works like this. You feed it a 4 bit number, and it connects the input line to a corresponding output line. So if we pass in 0000, it will select the very first column for us.

===== (08:00) to (10:00) =====

If we pass in 0001, the next column is selected, and so on. We need one multiplexer to handle our rows and another multiplexer to handle the columns. Ok, it's starting to get complicated again, so let's make our 256-bit memory its own component. Once again a new level of abstraction!

It takes an 8-bit address for input - the 4 bits for the column and 4 for the row. We also need write and read enable wires. And finally, we need just one data wire, which can be used to read or write data. Unfortunately, even 256-bits of memory isn't enough to run much of anything, so we need to scale up even more! We're going to put them in a row. Just like with the registers. We'll make a row of 8 of them, so we can store an 8 bit number - also known as a byte. To do this, we feed the exact same address into all 8 of our 256-bit memory components at the same time, and each one saves one bit of the number. That means the component we just made can store 256 bytes at 256 different addresses. Again, to keep things simple, we want to leave behind this inner complexity. Instead of thinking of this as a series of individual memory modules and circuits, we'll think of it as a uniform bank of addressable memory. We have 256 addresses, and at each address, we can read or write an 8-bit value. We're going to use this memory component next episode when we build our CPU.

The way that modern computers scale to megabytes and gigabytes of memory is by doing the same thing we've been doing here -- keep packaging up little bundles of memory into larger, and larger, and larger arrangements. As the number of memory locations grow, our addresses have to grow as well. 8 bits hold enough numbers to provide addresses for 256 bytes of our memory, but that's all. To address a gigabyte -- or a billion bytes of memory -- we need 32-bit addresses. An important property of this memory is that we can access any memory location, at any time, and in a random order. For this reason, it's called Random-Access Memory or RAM.

===== (10:00) to (12:00) =====

When you hear people talking about how much RAM a computer has - that's the computer's memory. RAM is like a human's short term or working memory, where you keep track of things going on right now - like whether or not you had lunch or paid your phone bill. Here's an actual stick of RAM - with 8 memory modules soldered onto the board. If we carefully opened up one of these modules and zoomed in, The first thing you would see are 32 squares of memory. Zoom into one of those squares, and we can see each one is comprised of 4 smaller blocks. If we zoom in again, we get down to the matrix of individual bits. This is a matrix of 128 by 64 bits. That's 8192 bits in total. Each of our 32 squares has 4 matrices, so that's 32 thousand, 7 hundred and 68 bits. And there are 32 squares in total. So all in all, that's roughly 1 million bits of memory in each chip. Our RAM stick has 8 of these chips, so in total, this RAM can store 8 million bits, otherwise known as 1

megabyte. That's not a lot of memory these days -- this is a RAM module from the 1980's. Today you can buy RAM that has a gigabyte or more of memory - that's billions of bytes of memory.

So, today, we built a piece of SRAM - Static Random-Access Memory -- which uses latches. There are other types of RAM, such as DRAM, Flash memory, and NVRAM. These are very similar in function to SRAM, but use different circuits to store the individual bits -- for example, using different logic gates, capacitors, charge traps, or memristors. But fundamentally, all of these technologies store bits of information in massively nested matrices of memory cells. Like many things in computing, the fundamental operation is relatively simple...it's the layers and layers of abstraction that's mind blowing -- like a Russian doll that keeps getting smaller and smaller and smaller. I'll see you next week.

Crash Course Computer Science is produced in association with PBS Digital Studios. At their channel you can check out a playlist of shows like Braincraft, Coma Niddy, and PBS Infinite Series. This episode was filmed at the Chad and Stacy Emigholz Studios in Indianapolis, Indiana.

===== (12:00) to (12:17) =====

It was made with the help of all of these nice people, and our wonderful graphics team Thought Cafe. Thanks for the random access memories, I'll see you next time.