



Boolean Logic & Logic Gates: Crash Course Computer Science #3

Crash Course: Computer Science

<https://youtube.com/watch?v=gl-qXk7XojA>

<https://nerdfighteria.info/v/gl-qXk7XojA>

Hi, I'm Carrie Anne and welcome to Crash Course Computer Science!

Today we start our journey up the ladder of abstraction, where we leave behind the simplicity of being able to see every switch and gear, but gain the ability to assemble increasingly complex systems.

[INTRO]

Last episode, we talked about how computers evolved from electromechanical devices, that often had decimal representations of numbers – like those represented by teeth on a gear – to electronic computers with transistors that can turn the flow of electricity on or off.

And fortunately, even with just two states of electricity, we can represent important information. We call this representation Binary -- which literally means "of two states," in the same way a bicycle has two wheels or a biped has two legs. You might think two states isn't a lot to work with, and you'd be right!

But, it's exactly what you need for representing the values "true" and "false." In computers, an "on" state, when electricity is flowing, represents true. The off state, no electricity flowing, represents false.

We can also write binary as 1's and 0's instead of true's and false's – they are just different expressions of the same signal – but we'll talk more about that in the next episode. Now it is actually possible to use transistors for more than just turning electrical current on and off, and to allow for different levels of current. Some early electronic computers were ternary, that's three states, and even quinary, using 5 states.

The problem is, the more intermediate states there are, the harder it is to keep them all separate. If your smartphone battery starts running low or there's electrical noise because someone's running a microwave nearby, the signals can get mixed up... and this problem only gets worse with transistors changing states millions of times per second! So, placing two signals as far apart as possible - using just 'on and off' - gives us the most distinct signal to minimize these issues.

Another reason computers use binary is that an entire branch of mathematics already existed that dealt exclusively with true and false values. And it had figured out all of the necessary rules and operations for manipulating them. It's called Boolean Algebra! George Boole, from which Boolean Algebra later got its name, was a self-taught English mathematician in the 1800s. He was interested in representing logical statements that went "under, over, and beyond" Aristotle's approach to logic, which was, unsurprisingly, grounded in philosophy. Boole's approach allowed truth to be systematically and formally proven, through logic equations which he introduced in his first book, "The Mathematical Analysis of Logic" in 1847.

In "regular" algebra -- the type you probably learned in high school -- the values of variables are numbers, and operations on those numbers are things like addition and multiplication. But in Boolean Algebra, the values of variables are true and false, and the operations are logical. There are three fundamental operations in Boolean Algebra: a NOT, an AND, and an OR operation. And these operations turn out to be really useful so we're going to look at them individually.

A NOT takes a single boolean value, either true or false, and negates it. It flips true to false, and false to true. We can write out a little logic table that shows the original value under Input, and the outcome after applying the operation under Output.

Now here's the cool part -- we can easily build boolean logic out of transistors. As we discussed last episode, transistors are really just little electrically controlled switches. They have three wires: two electrodes and one control wire.

When you apply electricity to the control wire, it lets current flow through from one electrode, through the transistor, to the other electrode. This is a lot like a spigot on a pipe -- open the tap, water flows, close the tap, water shuts off. You can think of the control wire as an input, and the wire coming from the bottom electrode as the output. So with a single transistor, we have one input and one output. If we turn the input on, the output is also on because the current can flow through it. If we turn the input off, the output is also off and the current can no longer pass through.

Or in boolean terms, when the input is true, the output is true. And when the input is false, the output is also false. Which again we can show on a logic table. This isn't a very exciting circuit though because it's not doing anything -- the input and output are the same. But, we can modify this circuit just a little bit to create a NOT. Instead of having the output wire at the end of the transistor, we can move it before. If we turn the input on, the transistor allows current to pass through it to the "ground", and the output wire won't receive that current - so it will be off.

In our water metaphor grounding would be like if all the water in your house was flowing out of a huge hose so there wasn't any water pressure left for your shower. So in this case if the input is on, output is off.

When we turn off the transistor, though, current is prevented from flowing down it to the ground, so instead, current flows through the output wire. So the input will be off and the output will be on. And this matches our logic table for NOT, so congrats, we just built a circuit that computes NOT! We call them NOT gates - we call them gates because they're controlling the path of our current.

The AND Boolean operation takes two inputs, but still has a single output. In this case the output is only true if both inputs are true. Think about it like telling the truth. You're only being completely honest if you don't lie even a little.

For example, let's take the statement, "My name is Carrie Anne AND I'm wearing a blue dress." Both of those facts are true, so the whole statement is true. But if I said, "My name is Carrie Anne AND I'm wearing pants" that would be false, because I'm not wearing pants. Or trousers. If you're in England.

The Carrie Anne part is true, but a true AND a false, is still false. If I were to reverse that statement it would still obviously be false, and if I were to tell you two complete lies that is also false, and again we can write all of these combinations out in a table.

To build an AND gate, we need two transistors connected together so we have our two inputs and one output. If we turn on just transistor A, current won't flow because the current is stopped by transistor B. Alternatively, if transistor B is on, but the transistor A is off, the same thing, the current can't get through. Only if transistor A AND transistor B are on does the output wire have current.

The last boolean operation is OR -- where only one input has to be true for the output to be true. For example, my name is Margaret Hamilton OR I'm wearing a blue dress. This is a true statement because although I'm not Margaret Hamilton unfortunately, I am wearing a blue dress, so the overall statement is true. An OR statement is also true if both facts are true. The only time an OR statement is false is if both inputs are false.

Building an OR gate from transistors needs a few extra wires.



Boolean Logic & Logic Gates: Crash Course Computer Science #3

Crash Course: Computer Science

<https://youtube.com/watch?v=gl-qXk7XojA>

<https://nerdfighteria.info/v/gl-qXk7XojA>

Instead of having two transistors in series -- one after the other -- we have them in parallel. We run wires from the current source to both transistors.

We use this little arc to note that the wires jump over one another and aren't connected, even though they look like they cross. If both transistors are turned off, the current is prevented from flowing to the output, so the output is also off. Now, if we turn on just Transistor A, current can flow to the output. Same thing if transistor A is off, but Transistor B is on. Basically if A OR B is on, the output is also on. Also, if both transistors are on, the output is still on.

Ok, now that we've got NOT, AND, and OR gates, and we can leave behind the constituent transistors and move up a layer of abstraction. The standard engineers use for these gates are a triangle with a dot for a NOT, a D for the AND, and a spaceship for the OR. Those aren't the official names, but that's how I like to think of them.

Representing them and thinking about them this way allows us to build even bigger components while keeping the overall complexity relatively the same - just remember that that mess of transistors and wires is still there. For example, another useful boolean operation in computation is called an Exclusive OR - or XOR for short. XOR is like a regular OR, but with one difference: if both inputs are true, the XOR is false.

The only time an XOR is true is when one input is true and the other input is false. It's like when you go out to dinner and your meal comes with a side salad OR a soup -- sadly, you can't have both! And building this from transistors is pretty confusing, but we can show how an XOR is created from our three basic boolean gates.

We know we have two inputs again -- A and B -- and one output. Let's start with an OR gate, since the logic table looks almost identical to an OR. There's only one problem - when A and B are true, the logic is different from OR, and we need to output "false."

To do this we need to add some additional gates. If we add an AND gate, and the input is true and true, the output will be true. This isn't what we want.

But if we add a NOT immediately after this will flip it to false. Okay, now if we add a final AND gate and send it that value along with the output of our original OR gate, the AND will take in "false" and "true", and since AND needs both values to be true, its output is false. That's the first row of our logic table.

If we work through the remaining input combinations, we can see this boolean logic circuit does implement an Exclusive OR. And XOR turns out to be a very useful component, and we'll get to it in another episode, so useful in fact engineers gave it its own symbol too -- an OR gate with a smile.

But most importantly, we can now put XOR into our metaphorical toolbox and not have to worry about the individual logic gates that make it up, or the transistors that make up those gates, or how electrons are flowing through a semiconductor. Moving up another layer of abstraction.

When computer engineers are designing processors, they rarely work at the transistor level, and instead work with much larger blocks, like logic gates, and even larger components made up of logic gates, which we'll discuss in future episodes. And even if you are a professional computer programmer, it's not often that you think about how the logic that you are programming is actually implemented in the physical world by these teeny tiny components.

We've also moved from thinking about raw electrical signals to our

first representation of data - true and false - and we've even gotten a little taste of computation. With just the logic gates in this episode, we could build a machine that evaluates complex logic statements, like if "Name is John Green AND after 5pm OR is Weekend AND near Pizza Hut", then "John will want pizza" equals true. And with that, I'm starving, I'll see you next week.

Crash Course Computer Science was produced in association with PBS Digital Studios. At their channel you can check out a playlist of shows like Gross Science, ACS Reactions, and The Art Assignment. This episode was filmed in the Chand & Stacey Emigholz Studio in Indianapolis Indiana, and it was made with the help of all these nice people and our wonderful graphics team, Thought Cafe. Thanks for watching and try turning it off and then back on again.