



How Computers Calculate - the ALU: Crash Course Computer Science #5

Crash Course: Computer Science

<https://youtube.com/watch?v=115ZMmrOfnA>

<https://nerdfighteria.info/v/115ZMmrOfnA>

===== Introduction (0:02) =====

Hi, I'm Carrie Ann and this is Crash Course Computer Science.

So last episode, we talked about how numbers can be represented in binary. Like 00101010 is 42 in decimal. Representing and storing numbers is an important function of a computer, but the real goal is computation or manipulating numbers in a structured and purposeful way, like adding two numbers together. These operations are handled by a computer's arithmetic and logic unit, but most people call it by its street name: the ALU. The ALU is the mathematical brain of a computer. When you understand an ALU's design and function you'll understand a fundamental part of modern computers. It is *the* thing that does all of the computation in a computer. So basically, everything uses it.

First though, look at this beauty. This is perhaps *the* most famous ALU ever: the Intel 74181. When it was released in 1970, it was the first complete ALU that fit entirely inside a single chip, which was a *huge* engineering feat at the time. So today we're going to take those brilliant logic gates we learned about last week to build a simple ALU circuit with much of the same functionality as that 74181. And over the next few episodes we'll use this to construct a computer from scratch, so it's going to get a little bit complicated, but I think you guys can handle it.

[music intro]

===== Arithmetic Unit (1:25) =====

An ALU is really two units in one. There's an arithmetic unit and a logic unit. Let's start with the arithmetic unit, which is responsible for handling all numerical operations in a computer, like addition and subtraction. It also does a bunch of other simple things, like add 1 to a number, which is called an increment operation, but we'll talk about those later.

Today we're going to focus on the pièce de résistance, the crème de la crème of operations, that underlines (?~1:49) almost everything else a computer does: adding two numbers together. We could build this circuit entirely out of individual transistors, but that would get confusing really fast. So instead, as we talked about in episode three, we can use a higher level of abstraction and build our components out of logic gates. (?~2:10) In this case AND, OR, NOT, and XOR gates.

The simplest adding circuit that we can build takes two binary digits and adds them together. So, we have two inputs, A and B, and one output, which is the sum of those two digits. Just to clarify, A, B and the output are all single bits. There are only four possible input combinations. The first three are:

- $0 + 0 = 0$
- $1 + 0 = 1$
- $0 + 1 = 1$

Remember that in binary, 1 is the same as true and 0 is the same as false. So this set of inputs exactly matches the boolean logic of an XOR gate and we can use it as our 1 bit adder. But the fourth input combination, $1 + 1$, is a special case. $1 + 1$ is obviously, but there's no 2 digit in binary. So as we talked about last episode the result is 0 and the 1 is carried to the next column. So the sum is really 10 in binary. Now, the output of our XOR gate is partially correct. $1 + 1$ outputs 0, but we need an extra output wire for that carry bit. The carry bit is only true when the inputs are 1 *and* 1, because that's the only time when the result is bigger than one bit can store. And conveniently, we have a gate for that: an AND gate,

which is only true when both inputs are true. So we'll add that to our circuit too. And that's it! This circuit is called a half adder. It's not that complicated, just two logic gates. But let's abstract away even this level of detail and encapsulate our nearly minted half adder as its own component, with two inputs, bits A and B, and two outputs, the SUM and the CARRY bits. This takes us to another level of abstraction... I feel like I've said that a lot, I wonder if this is gonna become a thing?

[music]

Anyway, if you want to add more than $1 + 1$, we're going to need a full adder. That half adder left us with a carry bit as output, that means that when we move on to the next column in a multi-column addition, and every column after that, we're going to have to add three bits together, not two. A full bit adder is more complicated. It takes three bits as inputs, A, B and C, so the maximum possible input is $1 + 1 + 1$, which equals 1 carry out 1. So we still only need two carry out wires, SUM and CARRY.

We can build a full adder using half adders. To do this, we use a half adder to add $A + B$, just like before, but then feed that result and input C into a second half adder. Lastly, we need an OR gate to check if either one of the carry bits was true. That's it! We just made a full adder! Again, we can go up a level of abstraction and wrap up this full adder as its own component. It takes three inputs, adds them, and outputs the SUM and the CARRY if there is one.

Armed with our new components, we can now build a circuit that takes two 8-bit numbers, let's call them A and B, and adds them together. Let's start with the very first bit of A and B, which we'll call A0 and B0. At this point, there is no carry bit to deal with, because this is our first addition. So we can use our half adder to add those two bits together. The output is SUM0. Now we want to add A1 and B1 together. It's possible there was a carry from the previous addition of A0 and B0, so this time we need to use a full adder that also inputs the carry bit. We output this result as SUM1. Then, we take any carry from this full adder and run it into the next full adder that handles A2 and B2. And we just keep doing this in a big chain so all 8 bits have been added. Notice how the carry bits ripple forward to each subsequent adder. For this reason, this is called an 8-bit ripple carry adder. Notice how our last full adder has a carry out. If there is a carry into the ninth bit, it means the sum of the two numbers is too large to fit into eight bits. This is called an overflow. In general, an overflow occurs when the result of an addition is too large to be represented by the number of bits you are using. This can usually cause errors and unexpected behavior.

Famously, the original PacMan arcade game used 8 bits to keep track of what level you were on. This meant that if you made it past level 255, the largest number storable in 8 bits, to level 256, the ALU overflowed. This caused a bunch of errors and glitches, making the level unbeatable. The bug became a right of passage for the greatest PacMan players. So if we want to avoid overflows, we can extend our circuit with more full adders, allowing us to add 16 or 32 bit numbers. This makes overflows less likely to happen, but at the expense of more gates. An additional downside is that it takes a little bit of time for each of the carries (?~6:18) to ripple forward. Admittedly, not very much time. Electrons move pretty fast, so we're talking about billionths of a second, but that's enough to make a difference in today's fast computers. For this reason, modern computers use a slightly different adding circuit called a carry-look-ahead adder, which is faster, but ultimately does exactly the same thing: adds binary numbers.

The ALU's arithmetic unit also has circuits for other math operations, and in general, these eight operations are always supported. And like our adder, these other operations are built from individual logic gates. Interestingly, you may have noticed that there



How Computers Calculate - the ALU: Crash Course Computer Science #5

Crash Course: Computer Science

<https://youtube.com/watch?v=115ZMmrOfnA>

<https://nerdfighteria.info/v/115ZMmrOfnA>

are no multiply and divide operations. That's because simple ALUs don't have a circuit for this, and instead just perform a series of additions. Let's say you want to multiply 12 by 5. That's the same thing as adding 12 to itself 5 times. So it would take five passes through the ALU to do this one multiplication. And this is how many simple processors, like those in your thermostat, TV remote, and microwave to multiplication. It's slow, but it gets the job done. However, fancier processors, like those in your laptop or smartphone have arithmetic units with dedicated circuits for multiplication, and as you might expect, the circuit is more complicated than addition. It's not magic, it just takes a lot more logic gates, which is why less expensive processors don't have this feature.

===== Logic Unit (7:32) =====

Okay, let's move on to the other half of the ALU, the logic unit. Instead of arithmetic operations, the logic unit performs, well, logical operations like AND, OR, and NOT, which we've talked about previously. It also performs simple numerical tests, like checking if a number is negative. For example, here's a circuit that tests if the output of the ALU is zero. It does this using a bunch of OR gates to see if any of the bits are 1. Even if one single bit is 1, we know the number can't be zero, and then we use a final NOT gate to flip this input so the output is 1 only if the input number is 0.

===== Wrap Up (8:05) =====

So that's a high level overview of what makes up an ALU. We even built several of the main components from scratch, like our ripple adder and you saw it's just a big bunch of logic gates connected in clever ways. Which brings us back to the ALU you admired so much at the beginning of the episode, the Intel 74181. Unlike the 8 bit ALU we made today, the 74181 could only handle 4 bit inputs, which means you build an ALU that's like twice as good as that super famous one! With your mind! Well, sort of. We didn't build the whole thing, but you get the idea. The 74181 used about 70 logic gates, and it couldn't multiply or divide, but it was a *huge* step forward in miniturization (?~8:44), opening the doors to more capable, less expensive computers. This 4 bit ALU circuit is already a lot to take in, but our 8 bit ALU would require hundreds of logic gates to fully build, and engineers didn't want to see all that complexity when using an ALU. So they came up with a special symbol to wrap it all up, which looks like a big V. Just another level of abstraction!

[music]

Our 8 bit ALU has two inputs, A and B, each with 8 bits. We also need a way to specify what operation the ALU should perform, addition or subtraction. For that, we use a 4 bit operation code, we'll talk more about this in a later episode, but in brief, 1000 might be the command to add, while 1100 is the command for subtract. Basically, the operation code tells the ALU what operation to perform. And the result of that operation on inputs A and B is an 8 bit output. ALUs also output a series of flags which, which are 1 bit outputs for particular states and statuses. For example, if we subtract two numbers and the result is 0, our 0 testing circuit (the one we made earlier) sets the zero flag to true. This is useful if we are trying to determine if two numbers are equal. If we wanted to test if A is less than B, we can use the ALU to calculate A subtract B and look to see if the negative flag is set to true. If it was, we know that A was smaller than B. And finally, there's also a wire attached to the carry out on the adder we built. So if there is an overflow, we'll know about it. This is called the overflow flag. Fancier ALUs will have more flags, but these three flags are

universal and frequently used. In fact, we'll be using them soon in a future episode.

So now you know your computer does all its basic mathematical operations digeously (?~10:25) with no gears or levers required. We're going to use this ALU when we construct our CPU two episodes from now. But before that, our computer is going to need some memory. We'll talk about that next week.

===== Acknowledgements (10:36) =====

Crash Course Computer Science was produced in association with PBS Digital Studios. At their channel you can check out a playlist of shows like PBS Idea Channel, Physics Girl, and It's OK to be Smart. 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 I'll CPU later.