



## Instructions & Programs: Crash Course Computer Science #8

Crash Course: Computer Science

<https://youtube.com/watch?v=zlTgXvg6r3k>

<https://nerdfighteria.info/v/zlTgXvg6r3k>

Hi, I'm Carrie Anne and this is Crash Course Computer Science!

Last episode, we combined an ALU, control unit, some memory, and a clock together to make a basic, but functional Central Processing Unit – or CPU – the beating, ticking heart of a computer. We've done all the hard work of building many of these components from the electronic circuits up, and now it's time to give our CPU some actual instructions to process!

The thing that makes a CPU powerful is the fact that it is programmable – if you write a different sequence of instructions, then the CPU will perform a different task. So the CPU is a piece of hardware which is controlled by easy-to-modify software! INTRO Let's quickly revisit the simple program that we stepped through last episode.

The computer memory looked like this. Each address contained 8 bits of data. For our hypothetical CPU, the first four bits specified the operation code, or opcode, and the second set of four bits specified an address or registers.

In memory address zero we have 0010 1110. Again, those first four bits are our opcode which corresponds to a "LOAD\_A" instruction. This instruction reads data from a location of memory specified in those last four bits of the instruction and saves it into Register A.

In this case, 1110, or 14 in decimal. So let's not think of this of memory address 0 as "0010 1110", but rather as the instruction 1:27"LOAD\_A 14". That's much easier to read and understand!

And for me to say! And we can do the same thing for the rest of the data in memory. In this case, our program is just four instructions long, and we've put some numbers into memory too, 3 and 14.

So now let's step through this program: First is LOAD\_A 14, which takes the value in address 14, which is the number 3, and stores it into Register A. Then we have a "LOAD\_B 15" instruction, which takes the value in memory location 15, which is the number 14, and saves it into Register B. Okay.

Easy enough. But now we have an "ADD" instruction. This tells the processor to use the ALU to add two registers together, in this case, B and A are specified.

The ordering is important, because the resulting sum is saved into the second register that's specified. So in this case, the resulting sum is saved into Register A. And finally, our last instruction is "STORE\_A 13", which instructs the CPU to write whatever value is in Register A into memory location 13.

Yesss! Our program adds two numbers together. That's about as exciting as it gets when we only have four instructions to play with.

So let's add some more! Now we've got a subtract function, which like ADD, specifies two registers to operate on. We've also got a fancy new instruction called JUMP.

As the name implies, this causes the program to "jump" to a new location. This is useful if we want to change the order of instructions, or choose to skip some instructions. For example, a JUMP 0, would cause the program to go back to the beginning.

At a low level, this is done by writing the value specified in the last four bits into the instruction address register, overwriting the current value. We've also added a special version of JUMP called JUMP\_NEGATIVE. This only jumps the program if the ALU's negative flag is set to true.

As we talked about in Episode 5, the negative flag is only set when

the result of an arithmetic operation is negative. If the result of the arithmetic was zero or positive, the negative flag would not be set. So the JUMP\_NEGATIVE won't jump anywhere, and the CPU will just continue on to the next instruction.

And finally, computers need to be told when to stop processing, so we need a HALT instruction. Our previous program really should have looked like this to be correct, otherwise the CPU would have just continued on after the STORE instruction, processing all those 0's. But there is no instruction with an opcode of 0, and so the computer would have crashed!

It's important to point out here that we're storing both instructions and data in the same memory. There is no difference fundamentally -- it's all just binary numbers. So the HALT instruction is really important because it allows us to separate the two.

Okay, so let's make our program a bit more interesting, by adding a JUMP. We'll also modify our two starting values in memory to 1 and 1. Let's step through this program just as our CPU would.

First, LOAD\_A 14 loads the value 1 into Register A. Next, LOAD\_B 15 loads the value 1 into Register B. As before, we ADD registers B and A together, with the sum going into Register A.  $1+1=2$ , so now Register A has the value 2 in it (stored in binary of course) Then the STORE instruction saves that into memory location 13.

Now we hit a "JUMP 2" instruction. This causes the processor to overwrite the value in the instruction address register, which is currently 4, with the new value, 2. Now, on the processor's next fetch cycle, we don't fetch HALT, instead we fetch the instruction at memory location 2, which is ADD B A.

We've jumped! Register A contains the value 2, and register B contains the value 1. So  $1+2=3$ , so now Register A has the value 3.

We store that into memory. And we've hit the JUMP again, back to ADD B A.  $1+3=4$ . So now register A has the value 4.

See what's happening here? Every loop, we're adding one. Its counting up!

Cooooool. But notice there's no way to ever escape. We're never.. ever.. going to get to that halt instruction, because we're always going to hit that JUMP.

This is called an infinite loop – a program that runs forever... ever... ever... ever... ever To break the loop, we need a conditional jump. A jump that only happens if a certain condition is met. Our JUMP\_NEGATIVE is one example of a conditional jump, but computers have other types too - like JUMP IF EQUAL and JUMP IF GREATER.

So let's make our code a little fancier and step through it. Just like before, the program starts by loading values from memory into registers A and B. In this example, the number 11 gets loaded into Register A, and 5 gets loaded into Register B.

Now we subtract register B from register A. That's 11 minus 5, which is 6, and so 6 gets saved into Register A. Now we hit our JUMP\_NEGATIVE.

The last ALU result was 6. That's a positive number, so the negative flag is false. That means the processor does not jump.

So we continue on to the next instruction... 5:51...which is a JUMP 2. No conditional on this one, so we jump to instruction 2 no matter what. Ok, so we're back at our SUBTRACT Register B from



## Instructions & Programs: Crash Course Computer Science #8

Crash Course: Computer Science

<https://youtube.com/watch?v=zlTgXvg6r3k>

<https://nerdfighteria.info/v/zlTgXvg6r3k>

Register A. 6 minus 5 equals 1.

So 1 gets saved into register A. Next instruction. We're back again at our JUMP NEGATIVE. 1 is also a positive number, so the CPU continues on to the JUMP 2, looping back around again to the SUBTRACT instruction.

This time is different though. 1 minus 5 is negative 4. And so the ALU sets its negative flag to true for the first time. Now, when we advance to the next instruction, JUMP\_NEGATIVE 5, the CPU executes the jump to memory location 5.

We're out of the infinite loop! Now we have a ADD B to A. Negative 4 plus 5, is positive 1, and we save that into Register A.

Next we have a STORE instruction that saves Register A into memory address 13. Lastly, we hit our HALT instruction and the computer rests. So even though this program is only 7 instructions long, the CPU ended up executing 13 instructions, and that's because it looped twice internally.

This code calculated the remainder if we divide 5 into 11, which is one. With a few extra lines of code, we could also keep track of how many loops we did, the count of which would be how many times 5 went into 11... we did two loops, so that means 5 goes into 11 two times... with a remainder of 1. And of course this code could work for any two numbers, which we can just change in memory to whatever we want: 7 and 81, 18 and 54, it doesn't matter -- that's the power of software!

Software also allowed us to do something our hardware could not. Remember, our ALU didn't have the functionality to divide two numbers, instead it's the program we made that gave us that functionality. And then other programs can use our divide program to do even fancier things.

And you know what that means. New levels of abstraction! So, our hypothetical CPU is very basic -- all of its instructions are 8 bits long, with the opcode occupying only the first four bits.

So even if we used every combination of 4 bits, our CPU would only be able to support a maximum of 16 different instructions. On top of that, several of our instructions used the last 4 bits to specify a memory location. But again, 4 bits can only encode 16 different values, meaning we can address a maximum of 16 memory locations - that's not a lot to work with.

For example, we couldn't even JUMP to location 17, because we literally can't fit the number 17 into 4 bits. For this reason, real, modern CPUs use two strategies. The most straightforward approach is just to have bigger instructions, with more bits, like 32 or 64 bits.

This is called the instruction length. Unsurprisingly. The second approach is to use variable length instructions.

For example, imagine a CPU that uses 8 bit opcodes. When the CPU sees an instruction that needs no extra values, like the HALT instruction, it can just execute it immediately. However, if it sees something like a JUMP instruction, it knows it must also fetch the address to jump to, which is saved immediately behind the JUMP instruction in memory.

This is called, logically enough, an Immediate Value. In such processor designs, instructions can be any number of bytes long, which makes the fetch cycle of the CPU a tad more complicated. Now, our example CPU and instruction set is hypothetical, designed to illustrate key working principles.

So I want to leave you with a real CPU example. In 1971, Intel released the 4004 processor. It was the first CPU put all into a single chip and paved the path to the Intel processors we know and love today.

It supported 46 instructions, shown here. Which was enough to build an entire working computer. And it used many of the instructions we've talked about like JUMP ADD SUBTRACT and LOAD.

It also uses 8-bit immediate values, like we just talked about, for things like JUMPs, in order to address more memory. And processors have come a long way since 1971. A modern computer processor, like an Intel Core i7, has thousands of different instructions and instruction variants, ranging from one to fifteen bytes long.

For example, there's over a dozens different opcodes just for variants of ADD! And this huge growth in instruction set size is due in large part to extra bells and whistles that have been added to processor designs overtime, which we'll talk about next episode. See you next week!