

CSC 134 5901

Module Three – Additional Notes

This lecture material is optional, and is intended to provide context as to why we are doing what we're doing.

Topics:

How does a computer actually work?

What is involved in programming as a profession?

How does a computer actually work?

If you're content with "a computer is a box that does what you tell it to do", then consider this information optional. For some people, though, understanding what's going on under the hood is very helpful in understanding the material, so we'll talk about it. Plus, it's interesting.

A computer at its most basic level is a device that takes binary data and converts it into other binary data. It doesn't have to have a CPU or a graphics card, and in fact the earliest computers were basically hand-wired collections of circuits. For the types of things a modern computer do, everything is handled by miniature **integrated circuits** printed onto a silicon wafer, and this is why computers are small enough to fit in your pocket instead of small enough to fit into a large basement.

Part 1 – Circuits and Bits

I'm not an electrical engineer, so I can only explain electronics at the level of a hobbyist. But a wire will be considered to either carry a current, or not, and we can consider this "on" and "off". (Sometimes it's called "high" or "low", if a circuit is transmitting 5v, then at 5v it's high, and at very low values it's low.)

I'll go ahead and describe "on" as 1 (one), and "off" as 0 (zero). This gives us **binary** data. Binary is a number system where instead of there being ten digits (zero through nine), there are two digits.

Just like we can create larger numbers by using multiple digits (such as \$99.99), it's possible to describe larger numbers by using more than one binary digit. ("bit" is short for "binary digit", so I'll use that.)

With one bit, you have two numbers, so you could describe something like "It's raining, or it isn't". As we'll see when using the **if** statement, often zero is called False and one is called True when we're working with a true/false statement.

Two bits gives you four possible numbers (zero through three). More bits, the more you can transmit at once. You can work with numbers that are larger than the number of bits you're working with, just like you could count to 100 by only using your fingers, as long as you had some way of keeping track of how many times you've counted ten on your fingers.

Okay, back to circuits. It's possible to build a circuit that when given one combination of bits will send out a different combination. Some examples are AND gates and OR gates. (Again, even though we're taking a class on C++, and not circuits, this AND / OR idea will come up when dealing with **if** statements.)

An AND gate has two wires going in, and one wire going out. It will send a 1 out if it receives a 1 from both inputs, otherwise it sends a 0.

An OR gate has two wires going in, and one wire going out. It will send a 1 out if either input reads 1, otherwise it sends a 0.

I'll skip ahead past a bunch of electrical engineering details and say that by combining various types of gates such as NAND gates (which is another type of gate like these), it's possible to build arbitrarily complex collections of gates that when you send them one signal, will send out any other signal.

The details of how this is done is usually using **transistors** because they have special properties that either transmit or resist (making it easy to send a 1 or a 0) depending on the circumstances.

Part 2 – Microprocessors and Assembly

Okay, so if we had a ton of time and money we could build a collection of circuitry that would do what we want. But **microprocessors** are cheap and plentiful, so we'll just buy one instead. The CPU in your computer or phone is an example of a microprocessor. Again, it's mostly just a collection of circuitry printed on a small silicon wafer.

An actual PC or phone you own right now is actually going to have a SoC – a "System on a Chip" – that has more in there than just a microprocessor, since it handles graphics and other things. So I'll talk about the 1980s instead.

A 1980s era personal computer used a specific microprocessor as its **Central Processing Unit (CPU)**. The 6502, which is still sold today for various uses, was the CPU used in the NES, Apple II, Commodore 64, and many other machines of the time. The IBM used an 8086, and Intel chips today are a distant descendant of the 8086. So on these old systems, we can use "microprocessor" and "CPU" interchangeably. I'll stick with saying CPU.

The computer had a lot more hardware in it than just the CPU, like sound chips and video chips to generate output to the TV your computer was hooked up to.

Let's say you were in the 1980s with just a Commodore 64. You've got two options for how to program the computer. When you turn it on, it boots up to a BASIC interpreter. You can type in some commands in BASIC, and the computer will do things, but you wouldn't be talking to the 6502 directly. The interpreter was a built-in program that would read the text you typed and then tell the CPU what to do based on what you'd typed. Like having an interpreter to translate for you live during a conversation, it was slower as a result.

Games or applications that you'd buy were typically much faster and could do a lot more than any BASIC program. This is because these programs were written to directly control the CPU. OK, but how?

Well, the 6502 has a specifications sheet that you can just go download right now as a PDF. (Of course in the 80s you'd instead write to them and wait for them to mail you one.) Remember, we said that a CPU is just a more complicated version of our collection of circuits, and what it does is turn one series of bits into another series of bits.

Here's a scan of one of those old spec sheets if you're curious:

http://archive.6502.org/datasheets/rockwell_r650x_r651x.pdf

The chip has a bunch of pins on it, and you would wire the pins up to different things. There's a playlist on Youtube by a user named "Ben Eater" where he takes a 6502 and just connects wires to it to send high/low voltage to the input pins and hooks up LEDs that light up when the output pins go high, which is as "bare bones" as you can work with a CPU.

Again I'll skip a lot of the details (Ben's videos go from "it lights up when I hook up a battery" to "it runs 'Hello World', it takes him a while). The CPU does stuff when you send it specific inputs. For example it has commands like "store this value in local memory" and "jump to a location if this value stored is not zero".

Writing code by simply looking up every instruction, and finding out the zeros and ones that make up the instruction, is possible. It's just very very slow.

In the linked PDF there's a page called "Instruction Set" and every instruction has a three letter code associated. (STA is "Store in Accumulator", BNE is "Branch if Not Equal to Zero", for instance.) Rather than remember that STA is 00011010 or whatever, people used programs called **assemblers** to convert their collection of letter codes into the **machine code** that the CPU expects.

As we saw in the first assignment, assembly code isn't much more readable, either. Here's "Hello World" in 6502 assembly (from the website Rosetta Code).

6502 Assembly

```
; goodbyeworld.s for C= 8-bit machines, ca65 assembler format.
; String printing limited to strings of 256 characters or less.

a_cr    = $0d          ; Carriage return.
bsout   = $ffd2        ; C64 KERNEL ROM, output a character to current device.
                        ; use $fded for Apple 2

        .code

        ldx #0          ; Starting index 0 in X register.
printnext:
        lda text,x      ; Get character from string.
        beq done        ; If we read a 0 we're done.
        jsr bsout       ; Output character.
        inx             ; Increment index to next character.
        bne printnext   ; Repeat if index doesn't overflow to 0.
done:
        rts             ; Return from subroutine.

        .rodata

text:
        .byte  "Hello world!", a_cr, 0
```

semicolons are comments. Commands like “ldx” and “bne” are assembly instructions. Items like “done:” are labels so that the program can jump to that spot.

Just like we don’t really know how **cout** is implemented, just that we can use it and it works, the line **jsr bsout** calls a routine hardcoded into the computer’s memory that actually does the work of putting a character on the screen.

So if we try to write pseudocode for what this program does, it’s :

```
// set things up
// load the “H” from the first character in the area called ‘text’
// call the routine at ffd2 to print that character
// is the next character a zero?
// if it is, we’re done
// if not, repeat
```

You have to admit writing **cout << “Hello World” << endl;** is a lot easier.

But... if you were to get a C++ compiler for a Commodore 64 (and they do exist, even though C++ came along much later), the binary code that it created when you compiled your “Hello World” would be very similar to the binary code created by the assembly code above.

This is one benefit of a **compiled** language like C++. Your code is converted into assembly which is then converted into machine code, so it's about as fast as if you had written the assembly yourself.

How does “programming” work as a profession?

So whether we tried to puzzle through how C++ code is converted into a series of 0s and 1s that the processor acts on, or we just take it as a given that it works, we still have the problem of how to actually do something **useful** with a computer using C++.

Here's one very desirable situation. We have the ability to write programs, and would like some money. Someone else has money, and would like a program to do something for them.

If we just start writing a program, and then hand it over to them, there's a good chance that they'll say something like “I don't know what to do with this” or “This doesn't do what I wanted”, and not satisfactorily complete the transaction by giving you money. We want to avoid that.

(There's also the case in which you want to write a program to do something for yourself, but it's a good idea to treat that the same way, except you don't pay yourself. Wearing one “hat” when you're in programmer mode and one “hat” in customer mode is a good way to keep your thoughts straight in that case.)

Let's list off a few ways this transaction could happen, and then we'll worry about the common ground all of them have, which is that you want your customer to actually be happy with the money they've spent.

Put the software on a storage device, put that in a box, and sell the box in a store. (I'm old enough that I remember when this was how most PC software was sold, but it's still somewhat relevant for game consoles)

Give the software away for free, and ask for donations (this was “shareware” in the 80s, nowadays a programmer might have a Patreon)

Sell the software, then sell a cosmetic addon like “horse armor” for an additional fee



Give the software away for free, let the user get frustrated, and have the screen show a convenient button that solves their frustration in exchange for a small cash payment

Okay, enough kidding around. Another common option:

Pay employees to develop software for you, don't sell it at all

In fact, a large amount of software development jobs fall into that latter category. Banks, insurance companies, many different industries have in-house software developers who create software that isn't sold at all, but is used as an internal resource. This is the type of job that most people who are "programmers" probably have.

There's also a large and growing number of people who have some other job in a given field, but don't consider themselves programmers. Instead, they write code to help themselves do their job. One example would be a researcher who writes Python code to analyze and display the data they're using for research projects. No code is exchanged for money at all.