Appendix A. Hardware and Software for Beginning Programmers

Some things make intuitive sense. Some we see in nature, and others are human inventions such as the wheel or pizza.

Others require more of a leap of faith. How does a television convert some invisible wiggles in the air into sounds and moving images?

A computer is one of these hard-to-accept ideas. How can you type something and get a machine to do what you want?

When I was learning to program, it was hard to find answers to some basic questions. For example: some books explain computer memory with the analogy of books on a library shelf. I wondered, if you *read from memory*, the analogy implies you're taking a book from the shelf. So, does that erase it from memory? Actually, no. It's more like getting a *copy* of the book from the shelf.

This appendix is a short review of computer hardware and software, if you're relatively new to programming. I try to explain the things that become "obvious" eventually but may be sticking points at the start.

Hardware

Caveman Computers

When the cavemen Og and Thog returned from hunting, they would each add a rock to their own pile for each mammoth they slew. But they couldn't do much with the piles, other than gain bragging rights if one was noticeably larger than the other.

Distant descendents of Og (Thog got stomped by a mammoth one day, trying to add to his pile) would learn to count, and write, and use an abacus.

But some leaps of imagination and technology were needed to get beyond these tools to the concept of a computer. The first necessary technology was electricity.

Electricity

Ben Franklin thought that electricity was a flow of some invisible fluid from a place with more fluid (*positive*) to a place with less (*negative*). He was right, but got the terms backwards. Electrons flow from his "negative" to "positive," but electrons weren't discovered until much later—too late to change the terminology. So, ever since we've needed to remember that electrons flow one way and *current* is defined as flowing the other way.

We're all familiar with natural electrical phenomena like static electricity and lightning. After people discovered how to push electrons through conducting wires to make electrical *circuits*, we got one step closer to making computers.

I used to think that electric current in a wire was caused by jazzed electrons doing laps around the track. It's actually quite different. Electrons jump from one atom to another. They behave a little like ball bearings in a tube (or tapioca balls in a bubble tea straw). When you push a ball at one end, it pushes its neighbor, and so on until the ball at the other end is pushed out. Although an average electron moves slowly (*drift speed* in a wire is only about three inches/hour), this almost-simultaneous bumping causes the generated electromagetic wave to propagate very quickly: 50 to 99% the speed of light, depending on the conductor.

Inventions

We still needed:

- A way to remember things
- A way to do stuff with the things that we remembered

One memory concept was a *switch*: something that's either on or off, and

stays as it is until something flips it to the other state. An electrical switch works by opening or closing a circuit, allowing electrons to flow or blocking them. We use switches all the time to control lights and other electrical devices. What was needed was a way to control the switch itself by electricity.

The earliest computers (and televisions) used vacuum tubes for this purpose, but these were big and often burned out. The single key invention that led to modern computers was the *transistor*: smaller, more efficient, and more reliable. The final key step was to make transistors much smaller and connect them in *integrated circuits*. For many years, computers got faster and ridiculously cheaper as they became smaller and smaller. Signals move faster when the components are closer together.

But there's a limit to how small we can stuff things together. This electron friskiness encounters *resistance*, which generates heat. We reached that lower limit more than 10 years ago, and manufacturers have compensated by putting multiple "chips" on the same board. This has increased the demand for *distributed computing*, which I discuss in a bit.

Regardless of these details, with these inventions we have been able to construct *computers*: machines that can remember things and so something with them.

An Idealized Computer

Real computers have lots of complex features. Let's focus on the essential parts.

A circuit "board" contains the CPU, memory, and wires connecting them to each other and to plugs for external devices.

The CPU

The *CPU* (Central Processing Unit), or "chip," does the actual "computing":

- Mathematical tasks like addition

- Comparing values

Memory and Caches

RAM (Random Access Memory) does the "remembering." It's fast, but *volatile* (loses its data if power is lost).

CPUs have been getting ever faster than memory, so computer designers have been adding *caches*: smaller, faster memory between the CPU and main memory. When your CPU tries to read some bytes from memory, it first tries the closest cache (called an *L1* cache), then the next (*L2*), and eventually to main RAM.

Storage

Because main memory loses its data, we also need *nonvolatile* storage. Such devices are cheaper than memory and hold much more data, but are also much slower.

The traditional storage method has been "spinning rust": *magnetic disks* (or *hard drives* or *HDD*) with movable read-write heads, a little like vinyl records and a stylus.

A hybrid technology called *SSD* (Solid State Drive) is made of semiconductors like RAM, but is nonvolatile like magnetic disks. Price and speed falls between the two.

Inputs

How do you get data into the computer? For people, the main choices are keyboards, mice, and touchscreens.

Outputs

People generally see computer output with displays and printers.

Relative Access Times

The amount of time it takes to get data to and from any of these components varies tremendously. This has big practical implications, For example, software needs to run in memory and access data there, but it also needs to store data safely on nonvolatile devices like disks. The problem is that disks are thousands of times slower, and networks are even slower. This means that programmers spend a lot of time trying to make the best trade-offs between speed and cost.

In Computer Latency at a Human Scale, David Jeppesen compares them. I've derived Table A-1 from his numbers and others. The last columns —Ratio, Relative Time (CPU = one second) and Relative Distance (CPU = one inch)—are easier for us to relate to than the specific timings.

Table A-1. Relative access times

Location	Time	Ratio	Relative Time	Relative Distance
CPU	0.4 ns	1	1 sec	1 in
L1 cache	0.9 ns	2	2 sec	2 in
L2 cache	2.8 ns	7	7 sec	7 in
L3 cache	28 ns	70	1 min	6 ft
RAM	100 ns	250	4 min	20 ft
SSD	100 µs	250,000	3 days	4 miles
Mag disk	10 ms	25,000,000	9 months	400 miles
Internet: SF→NY	65 ms	162,500,000	5 years	2,500 miles

It's a good thing that a CPU instruction actually takes less than a nanosecond instead of a whole second, or else you could have a baby in the time it takes to access a magnetic disk. Because disk and network times are so much slower than CPU and RAM, it helps to do as much work in memory as you can. And since the CPU itself is so much faster than RAM, it makes sense to keep data contiguous, so the bytes can be served by the faster (but smaller) caches closer to the CPU.

Software

Given all this computer hardware, how would we control it? First, we have both *instructions* (stuff that tells the CPU what to do) and *data* (in-

puts and outputs for the instructions). In the *stored-program computer*, everything could be treated as data, which simplified the design. But how do you represent instructions and data? What is it that you save in one place and process in another? The far-flung descendants of caveman Og wanted to know.

In the Beginning Was the Bit

Let's go back to the idea of a *switch*: something that maintains one of two values. These could be on or off, high or low voltage, positive or negative—just something that can be set, won't forget, and can later provide its value to anyone who asks. Integrated circuits gave us a way to integrate and connect billions of little switches into small chips.

If a switch can have just two values, it can be used to represent a *bit*, or binary digit. This could be treated as the tiny integers `0` and `1`, yes and no, true and false, or anything we want.

However, bits are too small for anything beyond `0` and `1`. How can we convince bits to represent bigger things?

For an answer, look at your fingers. We use only 10 digits (0 through 9) in our daily lives, but we make numbers much bigger than 9 by *positional notation*. If I add `1` to the number `38`, the `8` becomes a `9` and the whole value is now `39`. If I add another `1`, the `9` turns into a `0` and I *carry the one* to the left, incrementing the `3` to a `4` and getting the final number `40`. The far-right number is in the "one's column," the one to its left is the "ten's column," and so on to the left, multiplying by 10 each time. With three decimal digits, you can represent a thousand (10 * 10 * 10) numbers, from `000` to `999`.

We can use positional notation with bits to make larger collections of them. A *byte* has eight bits, with 2^8 (256) possible bit combinations. You can use a byte to store, for example, small integers `0` to `255` (you need to save room for a zero in positional notation).

A byte looks like eight bits in a row, each bit with a value of either `0` (or off, or false) or `1` (or on, or true). The bit on the far right is the *least sig-*

nificant, and the leftmost one is the *most significant*.

Machine Language

Each computer CPU is designed with an *instruction set* of bit patterns (also called *opcodes*) that it understands. Each opcode performs a certain function, with input values from one place and output values to another place. CPUs have special internal places called *registers* to store these opcodes and values.

Let's use an simplified computer that works only with bytes, and has four byte-sized registers called `A`, `B`, `C`, and `D`. Assume that:

- The command opcode goes into register `A`
- The command gets its byte inputs from registers `B` and `C`
- The command stores its byte result in register `D`

(Adding two bytes could *overflow* a single byte result, but I'm ignoring that here to show what happens where.)

Say that:

- Register `A` contains the opcode for *add two integers*: a decimal `1` (binary `00000001`).
- Register `B` has the decimal value `5` (binary `00000101`).
- Register `C` has the decimal value `3` (binary `00000011`).

The CPU sees that an instruction has arrived in register `A`. It decodes and runs that instruction, reading values from registers `B` and `C` and passing them to internal hardware circuits that can add bytes. When it's done, we should see the decimal value `8` (binary `00001000`) in register `D`.

The CPU does addition, and other mathematical functions, using registers in this way. It *decodes* the opcode and directs control to specific circuits within the CPU. It can also compare things, such as "Is the value in `B` larger than the value in `C`?" Importantly, it also *fetches* values from memory to CPU and *stores* values from CPU to memory.

The computer stores *programs* (machine-language instructions and data) in memory and handles feeding instructions and data to and from the CPU.

Assembler

It's hard to program in machine language. You have to get specify every bit perfectly, which is very time consuming. So, people came up with a slightly more readable level of languages called *assembly language*, or just *assembler*. These languages are specific to a CPU design and let you use things like variable names to define your instruction flow and data.

Higher-Level Languages

Assembler is still a painstaking endeavor, so people designed *higher-level languages* that were even easier for people to use. These languages would be translated into assembler by a program called a *compiler*, or run directly by an *interpreter*. Among the oldest of these languages are FORTRAN, LISP, and C—wildly different in design and intended use, but similar in their place in computer architecture.

In real jobs you tend to see distinct software "stacks":

Mainframe
: IBM, COBOL, FORTRAN, and others

Microsoft
: Windows, ASP, C#, SQL Server

JVM
: Java, Scala, Groovy

Open source
: Linux, languages(Python, PHP, Perl, C, C++, Go), databases (MySQL, PostgreSQL), web (apache, nginx)

Programmers tend to stay in one of these worlds, using the languages and

tools within it. Some technologies, such as TCP/IP and the web, allow intercommunication between stacks.

Operating Systems

Each innovation was built on those before it, and generally we don't know or care how the lower levels even work. Tools build tools to build even more tools, and we take them for granted.

Tha major operating systems are:

Windows (Microsoft)
: Commercial, many versions

macOS (Apple)
: Commercial

Linux
: Open source

Unix
: Many commercial versions, largely replaced by Linux

An operating system contains:

A kernel
: Schedules and controls programs and I/O

Device drivers
: Used by the kernel to access RAM, disk, and other devices

Libraries
: Source and binary files for use by developers

Applications
: Standalone programs

The same computer hardware can support more than one operating sys-

tem, but only one at a time. When an operating system starts up, it's called *booting*,[1] so *rebooting* is restarting it. These terms have even appeared in movie marketing, as studios "reboot" previous unsuccessful attempts. You can *dual-boot* your computer by installing more than one operating system, side by side, but only one can be fired up and run at a time.

If you see the phrase *bare metal*, it means a single computer running an operating system. In the next few sections, we step up from bare metal.

Virtual Machines

An operating system is sort of a big program, so eventually someone figured out how to run foreign operating systems as *virtual machines* (guest programs) on *host* machines. So you could have Microsoft Windows running on your PC, but fire up a Linux virtual machine atop it at the same time, without having to buy a second computer or dual-boot it.

Containers

A more recent idea is the *container*—a way to run multiple operating systems at the same time, as long as they share the same kernel. This idea was popularized by Docker, which took some little-known Linux kernel features and added useful management features. Their analogy to shipping containers (which revolutionized shipping and saved money for all of us) was clear and appealing. By releasing the code as open-source, Docker enabled containers to be adopted very quickly throughout the computer industry.

Google and other cloud providers had been quietly adding the underlying kernel support to Linux for years, and using containers in their data centers. Containers use fewer resources than virtual machines, letting you pack more programs into each physical computer box.

Distributed Computing and Networks

When businesses first started using personal computers, they needed

ways to make them talk to each other as well as to devices like printers. Proprietary networking software, such as Novell's, was originally used, but was eventually replaced by TCP/IP as the internet emerged in the mid- to late 90s. Microsoft grabbed its TCP/IP stack from a free Unix variant called *BSD*.[2]

One effect of the internet boom was a demand for *servers*: machines and software to run all those web, chat, and email services. The old style of *sysadmin* (system administration) was to install and manage all the hardware and software manually. Before long, it became clear to everyone that automation was needed. In 2006, Bill Baker at Microsoft came up with the *pets versus cattle* analogy for server management, and it has since become an industry meme (sometimes as *pets versus livestock*, to be more generic); see Table A-2.

Table A-2. Pets versus livestock

Pets	Livestock
Individually named	Automatically numbered
Customized care	Standardized
Nurse back to health	Replace

You'll often see, as a successor to "sysadmin," the term *DevOps*: development plus operations, a mixture of techniques to support rapid changes to services without blowing them up. Cloud services are extremely large and complex, and even the big companies like Amazon and Google have outages now and then.

The Cloud

People had been building computer *clusters* for a number of years, using many technologies. One early concept was a *Beowulf cluster*: identical commodity computers (Dell or something similar, instead of workstations like Sun or HP), linked by a local network.

The term *cloud computing* means using the computers in data centers to perform computing jobs and store data—but not just for the company that owned these backend resources. The services are provided to anyone, with fees based on CPU time, disk storage amounts, and so on. Amazon and its *AWS* (Amazon Web Services) is the most prominent, but *Azure* (Microsoft) and *Google Cloud* are also biggies.

Behind the scenes, these clouds use bare metal, virtual machines, and containers—all treated as livestock, not pets.

Kubernetes

Companies that needed to manage huge clusters of computers in many data centers—like Google, Amazon, and Facebook—have all borrowed or built solutions to help them scale:

Deployment
: How do you make new computing hardware and software available? How do you replace them when they fail?

Configuration
: How should these systems run? They need things like the names and addresses of other computers, passwords, and security settings.

Orchestration
: How do you manage all these computers, virtual machines, and containers? Can you scale up or down to adjust to load changes?

Service Discovery
: How do you find out who does what, and where it is?

Some competing solutions were built by Docker and others. But just in the past few years, it looks like the battle has been won by Kubernetes.

Google had developed large internal management frameworks, codenamed Borg and Omega. When employees brought up the idea of open sourcing these “crown jewels,” management had to think about it a bit,

but they took the leap. Google released Kubernetes version 1.0 in 2015, and its ecosystem and influence have grown ever since.

This refers to “Lifting yourself by your own

boot-
straps,”
which
seems
just
as
im-
prob-
a-
ble
as
a
com-
puter.

You
can
still
see
the
copy-
right
no-
tices
for
the
University
of
California
in
some
Microsoft
files.