It’s a valid question. I can see how it might seem that there is some sort of circular dependency or “chicken or the egg” situation. Nope.

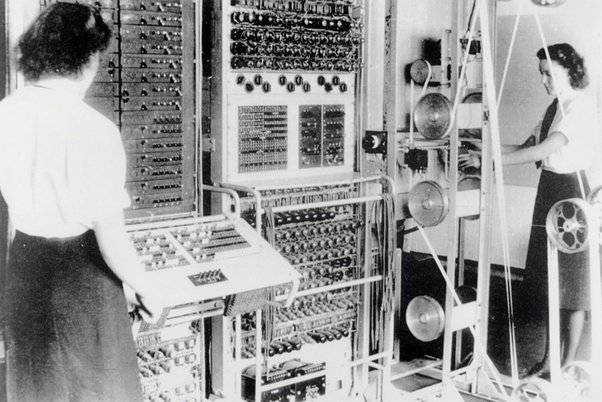
How exactly was it done then? I’ll try to explain but it’s going to take some time, diagrams, and pictures so please bear with me a bit.

**TL;DR down below *after the section break line*** (if you just want the gist of assembly language and creation of mnemonic instruction sets which lead to modern languages without the computer history).

In the beginning (1940’s and 1950’s) much of it was very manual. Early engineers actually flipped switches on panels or plugged cables into switched circuits. Later on they used punch cards where the holes in the cards would represent binary states of on or off (1 or 0). There were no mice, keyboards, or even screens originally (other than maybe an oscilloscope type screen in very early implementations). Things might be printed or show up on a manual read out or card device.

In time they came up with ideas to use these devices to further build and design other input devices, keyboards, mice, output devices such as monitors, printers, etc. It was a gradual building up of the year of technology before it and building onto it and so on. Eventually it was possible to have what we today consider a “modern computer”.

**Let’s look at a very early “computer” that was around circa 1945 for code breaking known as the Colossus Mark 2:**



Very manual. No real screen. Just inputting with cables and knobs, switches, and possibly a paper tape that reads out data or even just “prints” the result with a series of knobs that line up with the final computation (not unlike an abacus).

**Here is an IBM 650 Mainframe with IBM 533 punch card system (punch card deck is on the right - 650 operator console on the left). This is circa 1956:**



Not too much better. Still very manual and no real screen or anything resembling what we today would call a computer.

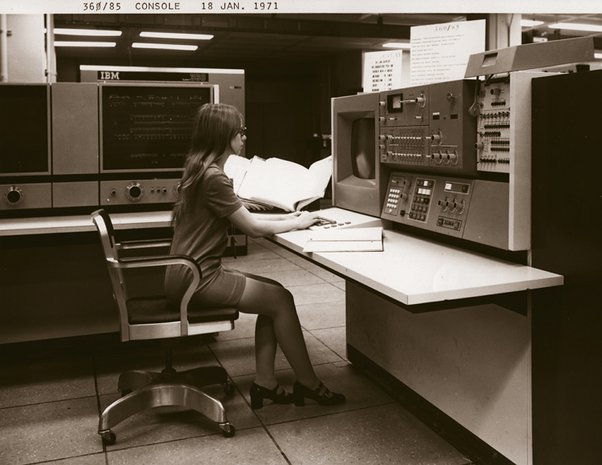
However internally it had many advancements, the ability to load punch card programs, operate much faster, a better instruction set, etc… Also the ability to add an IBM 653 storage unit which added magnetic memory storage, 60 10 digit words of magnetic core memory, 3 4 digit index registers, and floating point arithmetic precision. The instruction set was more advanced and had basic mnemonic features that assembly languages would later have.

**If we move up to circa 1968 we see something like the HP 9100:**



Hmm. This is beginning to look like a computer. Sort of. Technically it was called the first *scientific calculator* for marketing reasons but it’s basically one of the first desktop “computers”. It has a paper tape readout, a small screen, and a keyboard for inputting data. This was one of many of the first programmable calculators or personal “desktop” computers of the time.

**Circa 1971 here we have the IBM 360 mainframe and console:**



This is now beginning to look more like what we would consider a *computer*. There is a typical monitor for output and a typical keyboard for input. The mouse has not yet been invented.

**By 1981 we have IBM’s first official personal desktop computer running an Intel** processor:



**My point is that each generation builds on the previous. Languages developed as a way to program computers to do meaningful things. Their origins come from a laborious task of flipping switches over and over to get a desired effect (programming).**

**That being said, the typing we do today to program software, or what I'm doing right this instant, is just an *extremely* advancedand abstracted version of the same thing.** We consider it less laborious primarily due to: **1)** input devices today are incredibly smooth, light, quick, responsive and **2)** with a given instruction from our language of choice we can do phenomenally more, retrieve far more data, produce humongous amounts of output (relative to 2-5 decades ago) which is also a result of the fact that **3)** computers today are millions of times faster than the very best computers from even the 70s. In 1975 a CPU might handle .5-1 MIPS (Million Instructions Per Sec). Today we have desktops that can do **300,000 MIPS** (*300 billion instructions/sec*). That's 3-6 million times as fast as the *best* computers of 1975! It's an order of magnitude 100x that when compared to the very first computers of 1945-50. Mind blowing, hmm?

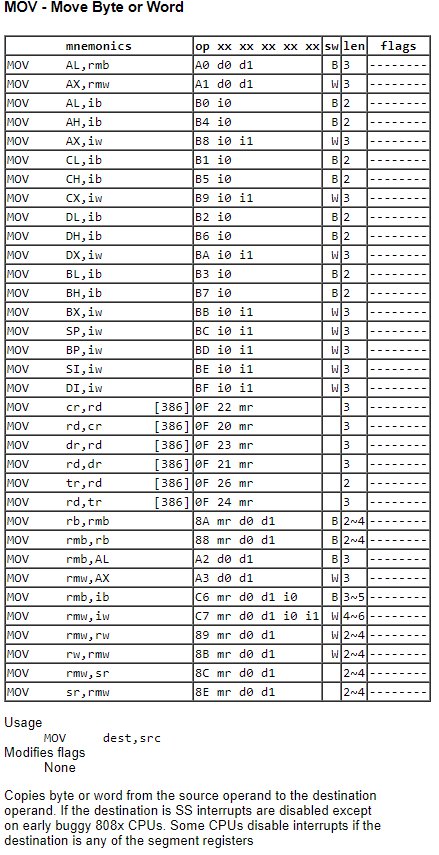
**Anyway, over time as these new devices came along and instruction sets (operations a CPU can carry out) became more advanced engineers created mnemonics to correlate to those instructions.**

Instead of a raw opcode they created systems where you could use a more human-friendly name, such as “MOV” (move register value) instead of having to remember an obscure binary or hex value like:

**Binary Opcode:** 00001010 00 01 00 00 11

**Hexadecimal Opcode**: A0 00 01 00 00 03

Instead you had something like this:



**That’s the MOV instructions mnemonic table for an Intel 8086 processor which moves data from a source location to a destination location.**

So, as time went on, transistor technology evolved, instruction sets became more complex, input devices were created to allow for easier programming, as well as output devices to easily see the results of the program, these mnemonics for various CPU opcodes and loading registers or memory locations with values were created.

We also created compilers and linkers that would allow you to write these programs using the mnemonics as in the diagram above. That’s what an assembler does. It translates what you have input, which is a mnemonic code, such as MOV (there are many others depending on the CPU instruction set), and it will convert it to the appropriate opcode with source and destination operand values in binary. That’s why we call assembler “machine” code, b/c there is more or less a one to one mapping for a mnemonic code to its equivalent binary instruction opcode and operands.

All higher level languages are just abstractions of assembler (i.e. they are complex assembly programs that abstract away the need for you to write in assembly). You can write the code in a more human readable format and the compiler will ultimately convert it to the appropriate assembly instructions (machine code). Operating systems that control the hardware, such as Windows, or Linux, are also just abstractions to allow us to use the device in a meaningful way. And they are also very much built on low level languages like assembly and C.

**They all build on the concepts from the previous generation. This is typically known as a form of “abstraction” or** [**bootstrapping**](https://en.wikipedia.org/wiki/Bootstrapping#Software_development)

**. Ultimately the way the first languages were created has to do with mnemonic codes that map to CPU instruction opcodes. That is how the first languages were created. Everything else just builds on those and *abstracts* those details away.**

A computer, or CPU, with a complete instruction set **is** a language. A binary only language.

**So in a fairly massive nutshell after all of the above explanation and pictures:**

* A computer or CPU ***is*** a programming language. It knows only binary. Binary is really just small electrical signals with 2 discrete states, switching on and off the various billions of transistors *logic gates* in a CPU. We use 1 and 0 to represent these 2 logical states but they are just small electric signals that will switch a gate on or off given a slight current.
  + ***Optional Reading:*** This is why we call transistors *semi*-conductors; silicon conducts a charge with just enough current. *Transistors* are essentially chemical etching onto wafers of silicon using a technique known as [Photolithography](https://en.wikipedia.org/wiki/Photolithography)

. The etching creates what today are literally billions of [logic gates](https://en.wikipedia.org/wiki/Boolean_algebra#Digital_logic_gates) which will make up the instructions of a CPU. The latest Skylake i7 Intel has around 5 billion transistors on a silicon die that is only 150*mm*2. They are created at the 14nm level using [FinFET (Field Effect Transistor)](https://en.wikipedia.org/wiki/FinFET). Each transistor is just 14 billionth’s of a meter wide. IBM & Samsung will have 4–5nm range models by 2020 commercially available. A single atom is about .1-.5nm wide ;) The gates are designed using [Boolean algebra](https://en.wikipedia.org/wiki/Boolean_algebra) which defines logic rules and [logic truth tables](https://en.wikipedia.org/wiki/Truth_table)

* + .
* An assembler had to be created for each CPU platform at some point. This would have been a simple language created in pure machine code. It’s main job would be to create a way to input values that were encoded as mnemonics (i.e. MOV, ADD, SUB…) and convert them to their binary opcodes.
* With a very raw and basic functioning assembler you could then create even better assemblers.
* With better assemblers programmers could then create even higher level abstractions (that I mentioned above) which are today our HLLs (high level languages) such as C, LISP, C++, Perl, etc…
* Eventually using many of those tools we made even more complicated languages such as .Net, C#, Java (these languages have a run time that compiles to machine language on the fly — known as JIT *just in time compilation*).

**It’s all, once again, abstraction upon abstraction upon abstraction for years and years until we have gotten to a point today where we rarely if ever need to rely on such low level interfaces to devices.** And many, except those very educated in the history and field of computer science, won’t realize how we got here. I wouldn’t be surprised in the slightest if the question came from a younger developer who is thinking about the origins and going, “huh?” — seriously.

63.4K views

View upvotes

View 49 shares

[Why are there so many programming languages?](https://www.quora.com/Why-are-there-so-many-programming-languages" \t "_top)

[— Related](https://www.quora.com/Why-are-there-so-many-programming-languages" \t "_top)

[[Profile photo for Dave Voorhis](https://www.quora.com/profile/Dave-Voorhis)](https://www.quora.com/profile/Dave-Voorhis" \t "_blank)

[Dave Voorhis](https://www.quora.com/profile/Dave-Voorhis" \t "_blank)

·

Follow

Software entrepreneur, engineer, and educator for 35+ years.Upvoted by

[Sverre Pippard Vigander](https://www.quora.com/profile/Sverre-Pippard-Vigander" \t "_blank)

, M.S. Computer Programming & Digital Electronics, Norwegian University of Science and Technology (2000) and

[Marcelo De Zen](https://www.quora.com/profile/Marcelo-De-Zen" \t "_blank)

, 15+ years as a software developer.[1y](https://www.quora.com/Why-do-so-many-different-programming-languages-exist/answer/Dave-Voorhis?no_redirect=1)

There was a time when every Computer Science Masters degree student, and many PhD students, would create new languages for their thesis projects.

There was a time when every serious developer thought he or she could do better than *<insert popular language here>* and created a new language to replace it.

There was a time when *sometimes* the only way to meet technical requirements — or the best way to meet requirements — was to create a new language to express their solution.

There was a time when every major software company — and many smaller ones — viewed a new computer language as a commercial opportunity, and created one or more new languages to sell or otherwise monetise.

That produced *a lot* of languages.

Naturally, only a few became popular and most are now long forgotten, but the result is that there are *still* a lot of languages out there, and many new ones still being made for the same reasons.

But notably, only a small handful are popular, and most of those are at least twenty years old.

79.2K views

View 797 upvotes

View 16 shares

Answer requested by

[Abdullah](https://www.quora.com/profile/Abdullah-841" \t "_blank)

[1 of 129 answers](https://www.quora.com/Why-do-so-many-different-programming-languages-exist?no_redirect=1" \t "_blank)

[[Profile photo for Tom Borkowski](https://www.quora.com/profile/Tom-Borkowski-1)](https://www.quora.com/profile/Tom-Borkowski-1" \t "_blank)

[Tom Borkowski](https://www.quora.com/profile/Tom-Borkowski-1" \t "_blank)

·

Follow

retired software manager studying grammar & human behavior[5y](https://www.quora.com/Why-are-there-so-many-programming-languages-that-do-the-same-thing/answer/Tom-Borkowski-1?no_redirect=1)

I and many other older software engineers and professors (the famous William Wesley Peterson for one) do NOT see the different programming languages as being essentially different. They all roughly do the same thing in pretty much the same manner, except for Functional Programming.

In the 60’s, the common belief was that different languages would develop for different industries, such as Fortran for science and COBOL for business. But it never worked out that way.

When The C Programming Language was developed, everyone began using it for everything except for the mainframe COBOL programmers.

Then the object mechanism was invented with C++, and then everyone switched to that for everything. We didn’t need new languages. C++ worked fine for everything. The object mechanism eliminated the need for new programming languages because new things could stored and then used as objects in a universal library.

Then JAVA can along which introduced the runtime environment which was a major development but the JAVA language itself was NOT essentially different from C++.

Then everyone developed their own language version of JAVA/C++ which didn’t invent anything new. It just totally messed up the computer programming industry and created the myth of the goodness of new languages. We do not need a hundred languages that all work the same way except for the syntax differences. Multiple languages only cause misery for computer programmers and the computer programming industry.

I personally like C#.NET which cleaned up that dirty C++ and standardized on a good runtime environment. But so did JAVA. Either one would be fine as the standard general-purpose programming language.

I wish that Microsoft, Google, and Oracle could cooperate to produce one standard general-purpose language and one easy-to-use development system because Visual Studio (and those other development systems) are a nightmare of idiosyncrasies and difficulty.

Instead of creating new languages, I wish our industry would concentrate on creating a universal library of objects maintained by the ACM or IEEE. Such a universal library of objects would drastically improve the capability of computer programmers because we would have access to all known algorithms and techniques instead of each of us reinventing the wheel over and over again.

Finally, I wish the myth of the goodness of different languages would go away. They are all essentially the same. It’s only the libraries, runtime environments, idiosyncrasies, and development systems that come with the different languages that are different. The languages themselves are essentially the same, but only with different syntaxes.

When I read a newbie asking which language should he learn, I want to scream. They’re all the same except for Functional Programming.

It’s the other programming subjects that the newbie should be studying such as data structures, algorithms, operating systems, numerical analysis, discrete mathematics, graph theory, and optimization—not ten different languages which all do the same thing but only with different syntaxes, libraries, development systems, idiosyncrasies, and runtime environments.

Can you imagine a medical doctor spending his college career studying 10 different human languages instead of taking medical courses?

The programming languages they are made for specific purpose