A Rant About CPUs, Languages, and Development Methods

My first computer was a *ZX81*, released in 1981. It had a *Zilog Z80* CPU and could be programmed in *Basic*. It was glacially slow, but it had a fast mode: The display was driven by the CPU, so turning it off increased the speed a lot, relatively speaking. It was a brilliant piece of engineering: There were a total of three chips on the main board including the CPU and not counting RAM, which was one or two additional chips. Other home computers at the time were built up of around twenty chips. So, it was cheap to produce, and semi-fun. Here is a picture of it. The *Weighted Companion Cube* is there as a size reference. It is pretty small:

Excerpt from the book *Real Programming* (2021) by Corneliusen & Julin.
More information: www.ignorantus.com/books/real/

Then I moved on to the *Commodore 64*. There might have been a *VIC-20* before that: My brother had one and we had some fun with it, but 3.5 KB RAM only gets you that far. The Commodore 64 had a whopping 64 KB RAM and good games, but it was still 8-bit. It used a *MOS Technology 6502* variant called the *6510*. Programming it was slightly less hard than the Z80, but only slightly. Basic moved along at a slightly less glacial speed. There were some custom video and audio chips in it, and the cool kids made, and still make, great stuff for it. I was thoroughly impressed by what they did with it back then. I never figured it out: Wrote some machine code in a terrible tool, but there was too much black magic involved.

A CPU executes instructions, one after another. This basic principle has not changed much in fifty years. Granted, there have been some advances in later years, but that is the gist of it. Obviously, a CPU can have more than one execution unit, and it can be pipelined and all that, but the fact remains: The ordering of the instructions in memory remain constant, and the CPU needs to decode them in order. It can reorder, of course, but there still is an optimal order to put them in. And how much reordering can be done at high clock speeds? Not a lot. There will always be a sensible order: Start loads as early as possible, use results as late as possible, and use all the registers if possible. It is not rocket science. Or sometimes that is just what it is, both for the people writing rocket control code, and for the people trying to squeeze the last remains of computational power out of a cheap microcontroller.

CPUs deal efficiently with basic data types. They have never had, and will probably never have, any concept of functions, methods, classes, untyped data, or other abstractions. There is one exception: *Intel* released such a CPU in 1981: The Intel *iAPX 432*. It failed miserably, mainly because it was horrendously slow, and the compilers were horrendously bad. Not a winning combination then, and not a winning combination now.

So why do programming languages converge towards such concepts? This is confusing for a programmer that enjoys writing efficient code. In my opinion, fast execution should always be the focus, especially now that people execute software on hosted servers and pay for every CPU second used. This might be a conspiracy by the server providers, who are forcing the customers to use the slow programming languages that require much more CPU than strictly needed. But that is probably far-fetched.

It is quite likely rooted in the invention of object-oriented languages in the sixties. They did not see widespread popularity until the mid-nineties, when CPUs started to become faster and useful for most people. The Internet caught on, more people had computers and not just the hackers wanted to write programs. There was always the hope that the next generation of CPUs would run the code better, and make real programming languages, like *C*, unnecessary.

In my polarized view, there have been two great revolutions in the CPU world: The launch of the *Motorola 68000* and, much later, the *Tilera TILE-Gx*. And they did the same as all the other CPUs, only better. Much better. And they still executed instructions one at a time and had no concept of the previously mentioned abstractions that have been so popular since the mid-nineties.

A fundamental problem back in the seventies was that the CPUs were not wide enough. Or fast enough. Or had enough memory. Or had fast enough memory. The point should be clear as bottled drinking water. The Z80 and 6502 and other CPUs back then were hugely successful in the booming home computer market, but they operated on 8-bit data types. Wheels had to be invented to add or multiply 32-bit numbers. Let skip the subject of division. The CPUs did not have all the necessary basic arithmetic operations for writing quick code in a civilized manner. So, while there was widespread adoption of home computers, they were rarely used for anything useful, except by the hackers of the time. I recall a program for the Commodore 64 that could keep track of the contents of your freezer. Great idea, except it took a couple of minutes to load from tape. Fun times!

As all the cool kids remember, Prince launched the album *Prince* in 1979, and Motorola launched the 68000 CPU. Let us focus on the latter. It should also be mentioned that Pink Floyd Released the album *The Wall* that year. 1979 was a momentous year, both in music and CPUs.

Anyway, the 68000 was a huge leap forward for us hobbyists, and afterwards the rest of the world. It can be considered the first 32-bit CPU. It was not really, but for most problem solving, it was. It had a whopping number of registers: 8 32-bit data registers and 8 32-bit address registers. Memory accesses used 32-bit pointers, unheard of in the industry so far. Only 24 bits were in use, but the registers were 32. Multiply was 16-bit times 16-bit and gave a 32-bit answer. There was also a division instruction that gave both the quotient and the rest as result. Division was not particularly fast, but with tools like that, it was possible to write real programs.

And it was *big-endian*. *Little-endian* was smart on 8-bit CPUs, since it made adding wider numbers directly from memory simpler: Wider words are byte mirrored, so the low 8 bits come first. On a 32-bit CPU it does not matter, and big-endian is the logical way to work with actual bits inside a word wider than 8 bits: A big-endian memory dump makes much more sense than a little-endian memory dump, at least to the people who still know what a memory dump is. I do not know why newer CPUs use little-endian. Bad habits are hard to (byte) reverse.

Several popular computer platforms used the 68000 CPU. The greatest one was, of course, the Commodore *Amiga*, but I am certain that will meet a lot of protests from *Atari ST* and Apple *Macintosh* fans. The Amiga had the best custom chips for graphics and sound and I/O at the time, putting it ahead of the competition.

Newer revisions of the 68000 added true 32-bit addressing, true 32-bit ALU and other modern features. It was an amazing chip to learn real programming on.

And what did we write code in? 68000 assembler for a long time. Then some half-decent C compilers started appearing for the Amiga, so we learned C. C is one of the few languages that puts as little fuzz as possible between what you write and the code that is generated. It has some quirks, but if you do not write crap, you do not get crap. It gave you mostly free hands, and if the result was not fast enough, it was simple to add some assembler code to the program.

Of course, the 68000 line did not stick. Intel finally started making a useful CPU that was 32-bit: The Intel *80386*, and Microsoft made an OS that supported it. It took a long time: While the '386 was launched in 1985, Microsoft did not get their OS act together until 1993 with the launch of *Windows NT*. For the public, it did not happen until 1995 with the launch of *Windows 95*.

As mentioned, object-oriented programming, a concept invented during the sixties, never really caught on until the middle of the nineties, when CPUs started getting a lot faster and more efficient. There were many completely different CPUs on the market: *Digital Alpha* was pretty great, there was *Sun's SPARC*, *MIPS* IV and *PowerPC* were good choices, and there were plenty more, in addition to Motorola and Intel, of course. It is funny how that turned out in the end (now, in 2020): There are mainly two architectures that are used, *ARM* and *x86*. Two. So, when trying to write fast code, that is the number of targets to think about. Not a lot.

Since CPU speeds were increasing rapidly, languages that were completely different from C were somehow necessary. The code was "compiled" to an intermediate language that could be emulated on the target CPU. Later, seeing as emulating that code was really slow, attempts were made to make native code from the intermediate code. A trend that continues to this day. It sucked then and it sucks now. It will continue sucking in the future.

Unfortunately, these languages caught on. They added craziness like dynamic allocation that needed garbage collectors, abstracted away basic data types, and basically ruined efficient programming. The hope was that the next faster CPU would run the so-called code quickly.

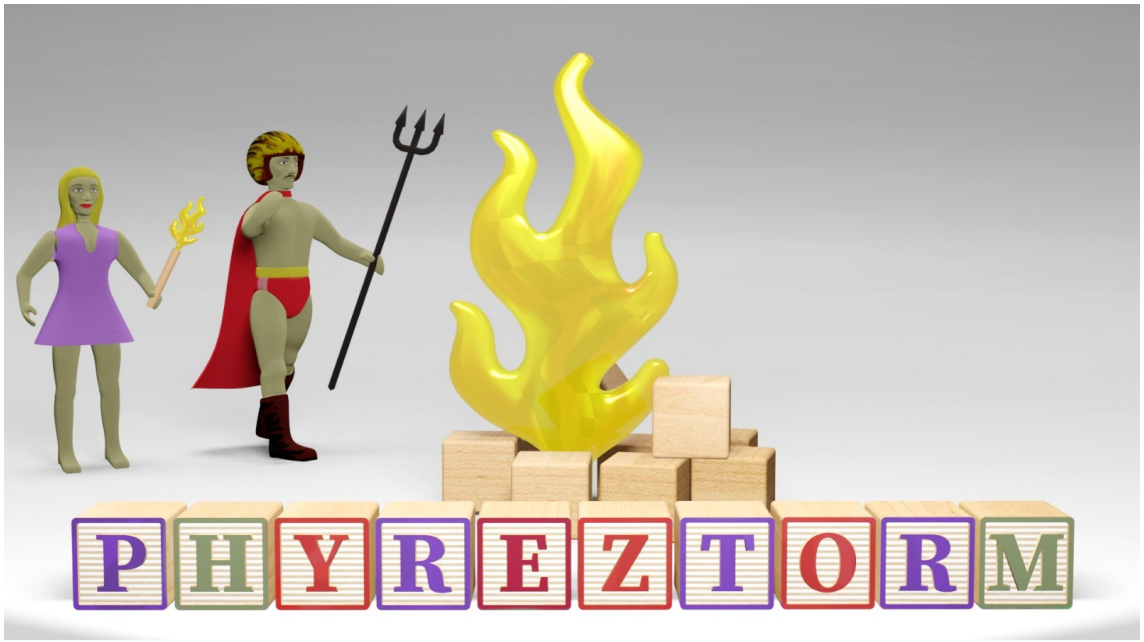
Some CPU manufacturing companies thought that integrating parts of the intermediate code execution into the CPU would make it better. ARM was one of them, with a technology that could execute some of the instructions, known as *bytecode*, directly in hardware. Unfortunately, the technology was locked down, so the open-source community shied away from it. It is now dead in the water. It was followed up with less locked-down versions, but I really have not heard anything about it since early 2000.

Unfortunately, Intel also hit a dead end with the *Netburst* architecture, and the *Pentium 4* CPUs introduced in the year 2000. These used an extremely deep pipeline to squeeze out a few extra instructions per Mhz. This worked well until the program hit a branch, in which the processor stalled - it took ages to clear the pipeline! They were stuck with this architecture until 2006, even though it severely underperformed. AMD managed to keep the train rolling a while with *Sledgehammer* and friends, before Intel retook the performance crown with their *Core* architecture in 2006. The Core architecture was one of the great successes of Intel and it was based on the Pentium M and Pentium III architectures that had lower clock speeds and were developed specifically to save power.

And in all this madness, the weird languages flourished. There was no longer a requirement to be an expert programmer or understand pointers to write programs! Some of these languages did not even have types. A turning point for me was the day I understood why such languages need the triple equality operator: Since nothing has a type, how is it possible to decide if two objects are explicitly the same, not just evaluating to the same? Right. That led to a firestorm of interesting bugs. I felt a moment of illumination, followed by sadness. I walked away, like every sane person would do.

I had that same feeling some years earlier when I understood the fad known as functional programming: Everything is evaluated as functions returning values. I have news for those who are into that: Modern CPUs do not work that way. Newer implementations have made that class of languages slightly more efficient, but I will claim they started in the wrong end. This piece of advice is free for everybody feeling the urge to design a new language: Find out how a CPU works first, and then design the language. Hint: Most CPUs work in the same way.

Let us hereafter refer to such languages as *toy languages*, with *Phyreztorm* being the name of such a language. It does not exist, of course.



I can hear them shouting: "Just write it in Phyreztorm and it will soon be fast! Really soon." I counted approximately two new programming languages on *Slashdot* every week in the years before the dot com bubble burst in 2000. They tended to solve one problem well. Each. And it started a trend that continues to this day: Every time new versions of these toy languages are published, all the code must be rewritten, since they had to fix the glaring errors in the previous version. Great stuff!

But here is the crux of the matter: Are those languages really any different? Do they introduce any revolutionary new concepts that will run fast? That would be the ultimate test, and so far, the only answer I see is "no". The invention factor is zero. Instead of converging toward what would be expected, fast code, they are converging towards the esoteric and theoretical, the prime example being Microsoft's *Bosque* that eliminates "unnecessary" constructs like loops. Really. I claim that most, if not all, of these languages are just regurgitations of other such languages. I will get back to the efficiency question later.

In modern days where companies rent CPU time from large server farms, it begs the question: Why will they not use more efficient code that would cost less? And for those that care about the environment, the answer should be obvious. Here is a case where efficiency and environmentalism align. That does not happen very often since they are completely unrelated. It is hilarious and a bit sad.

Following in that same trend, why are the smartphone companies not interested in efficient code? If most of what is running on them is slow, emulated code, that would surely drain the battery quicker? Why is there no interest in prolonging battery life? I remember driving up into the mountains with some friends a couple of years ago. Three hours after takeoff they started frantically searching for places to connect their chargers. Some even had extra battery packs shaped like lumps. A simple answer is that charging reduces the lifetime of the battery, so more charging leads to increased phone sales. But I try to avoid the conspiracy theories, even though it is not easy.

As mentioned, AMD managed to make something great in this period, around 2003: 64-bit extensions for the x86 instruction set dubbed *x64*. (There was a bewildering array of names for it. Let us stick with x64.) We were closing in on the 64-bit age! No more of this old 32-bit crap. If there is one truth in computing, it is that everything gets old and worn after a while. But in contrast to toy programming languages, the good CPU concepts live on: If you had a handle on 32-bit addressing, 64-bit would be easy.

But you know how the story goes. Otherwise, I suggest finding something else to read. I want my registers to be at least 64 bits and I want many of them. And I want to work on smaller sizes in parallel. *SIMD* (Single Instruction Multiple Data) solved some of these problems. There was a flurry of Intel/AMD SIMD extensions, starting with *MMX* in 1997 and followed by a list of spinoffs over the years. Some of the more popular ones were *SSE, SSE2, SS**S**E3, AVX, AVX2,* and *AVX-512*. MMX was 64-bit, SSE-variants 128-bit, AVX/AVX2 256-bit, and AVX-512 512-bit.

Unfortunately, they did not know when to stop. Writing efficient code with 512-bit registers is troublesome when there are only 32 of them. And the orthogonality of the instruction sets was and is insufficient. The peak was 128-bit. I will say that again: The peak was 128-bit. Instead of goofing around with 512-bit units, there should have been more 128-bit units. Or better yet: More SIMD registers. In 128-bit SSE2 mode there are only 16. Give me 32 and I will be happy. A good name will be *Mega-SSE2*.

Recently, in the summer of 2020, Intel has made announcements about a new architecture called *Lakefield*. While details are still thin, scarce really, it is a step in the right direction. As far as I can tell, the larger width SIMD units (AVX, AVX2, AVX-512) have been retired. It will be interesting to see how this ends up when actual chips are available.

VLIW (Very Long Instruction Word) was cool when it arrived. They were usually *DSPs*, meaning without all the security features in CPUs from Intel and the like, but they still deserve a mention for being on the edge. The basic idea was that several instructions were issued and sent to different execution units at the same time, as specified by the programmer.

The technology had been in the background for a long time, but in 1996 *Philips* released the *TriMedia TM1000*, the first such CPU I think was really useful. It had 5 execution pipelines and encoded 5 instructions as one (I am simplifying a lot here), so if all slots could not be filled, tough luck. Registers were 32-bit and plentiful: 128 of them. Every operation had a *conditional guard*, so it was possible to merge *if* branches, if that was the quickest way to do it. It had different operations per slot: All 5 had an ALU, all 5 had shifters, 2 had *DSPMUL* a.k.a. the cool SIMD operations, and several other specificities. Every operation was pipelined, except floating-point square root and division, but it had them if needed. It kicked ass!

The compiler had a rough time generating useful 5-slot bundles and exploiting the 2 SIMD slots, but it had intrinsic function support, also known as *intrinsics*: Write assembler instructions directly into the program like function calls. One of the greatest concepts ever. The compiler handled the register allocation and instruction ordering, and it usually did an excellent job, except for the few cases where the equally excellent assembler could be used. It saw some success, but later faded away and ended its life as an integrated controller in TVs. A disappointing end for an innovative architecture.

Texas Instruments tried launching an 8-way VLIW chip back in 2001 called the *TMS320C6x* family but fumbled it by just duct-taping 2 4-way cores together. Let us ignore that and keep our heads high. It was a success, but it was difficult to program. As there were 2 4-way cores, there were also two register banks, and the data needed were always in the wrong register bank. I liked VLIW until I tried writing code for the TI chip. Gave it up. The TI chip, that is.

I spent several years hacking away on the TriMedia, but the embedded world converged towards larger *SoCs: System on Chip*. SoCs are CPUs with all necessary I/O, and sometimes memory, in a single package. Sometimes it means there is an extra CPU in there handling the I/O. And that CPU runs software. And software has bugs. What a shocker! I remember doing some programming on an AMD *Elan SC400* SoC back in 1998, ran into some issues with the DMA controller, and consulted the local guru. He gave me a copy of the 600-page errata. Obviously, the errata did not describe that issue. Spent a week figuring it out and submitted a bug report. It was added in the next release of the errata, now 601 pages. Business as usual.

Another CPU worth mentioning is the one used in the *PlayStation 3* games console: *Cell*. It was launched to much fanfare in 2006. It had a PowerPC core for normal processing and 8 compute cores. They all had long, complicated names. Not important. The compute cores each had their own RAM and registers and could do small jobs blazingly fast. They were connected to a ring bus, so it could send blocks of data quickly to its neighbors. An innovative idea, but it was never really exploited to its full potential. It saw some use in supercomputers at the time but faded away. Well, they sold 87.4 million units of the PS3, so it was a success, I guess. I liked the idea, and they were fun to write code for. I just wanted more of those cores. Many more.

And guess what? The idea has seen a resurgence lately, in mega-chips made by *Graphcore*. In 2018 they launched their first chip, aptly called *Colossus*. It has a substantial number of Cell-like cores per chip that can be connected to other such chips via a high-speed network. They try to pitch it as a solution for modern AI and machine learning

problems. I would like to try something different: Let us try improving on normal problems instead. Compared to Cell, it should be able to blaze through large data sets in new and more efficient manners. Unfortunately, my supply of Graphcore chips is currently zero. I did apply for a job there once or twice, depending on who is counting. Did not work out. I do not blame them. But surely it would be fun to have such a chip to try out some alternative solutions. And I mean "fun". Occasionally, such fun results in actual new results. And sometimes it does not, which is a very tough sell to your local software manager. I will get back to that later.

The computing world rarely stands still, and it did not in this period either: Around 2009 the GPU guys went on a slimming spree, making their chips thinner and faster and adding more cores. The CPU guys decided to bulk up with mega-wide SIMD variants, the fattest one being Intel's *Xeon Phi* launched in 2013. It had AVX-512: 512-bit execution units and 32 registers. Each core had an *Atom* control core which could not manage to keep up with decoding, so there had to be multiple threads, via *Hyper-threading*, just to keep the mega-SIMD units busy. Seriously?

Hyper-threading launched back in 2002 and the idea was good: Switch to another thread quickly if there is a real stall, like a memory stall. Not instruction decode stall. I will claim that those are proof of a bad design. But I remember when Hyper-threading arrived: I first thought "oh great, that will increase the throughput" and then "oh crap, what about the security?". Well, that has received some attention lately and has been widely covered in the tech press, so no need to go into details about that.

ARM tried their best with *Neon* SIMD extensions from around 2005. Unfortunately, like SSE2 it suffers from not enough registers: In the old Neon there are 32 64-bit registers that can be called D registers, or they can be paired up as 16 128-bit Q registers, if they are next to each other and the first one has an even number. Right. Luckily, they cleaned up their register mess in *AArch64* Neon launched in 2011 with 32 128-bit registers. Thank you, ARM! And they cleaned up a lot of quirks in the instruction set, too. Neon does not feel tacked on anymore. Question to Intel: Can I get SSE2 with 32 registers, please? But what runs fast on the *NVidia Tegra X1 or X2* Neon unit might not run as quick on somebody else's. And, as is well known, nothing runs fast on the NVidia *Denver* cores that use some weird dynamic recompilation stuff. It is about as bad as it sounds. I will bury that corpse later.

What was needed was a revolutionary 64-bit CPU. Preferably with fewer security issues. As mentioned, in 1979, Prince and Pink Floyd revolutionized the music world (well, mainly Pink Floyd) and Motorola the computing world. In 2010, the only remarkable event in the media industry was the release of the movie *Kick-Ass*, so it was a pretty slow year. And it is a perfectly valid point that the movies, music, and games I like rarely align with the rest of the world. But there were remarkable things happening in the CPU market: Tilera's TILE-Gx CPU was launched. It was the first breath of fresh 64-bit air in a long time.

Tilera was known for making chips with lots of cores: They had launched the *TILE64* 64-core CPU in 2007, and the *TilePro* 64-core CPU in 2008. I did some hacks on them around 2009 but felt something was missing. Then the TILE-Gx appeared, and it had it all. Almost. There were: 64 64-bit registers, 3 execution pipelines (all instructions were 64-bit bundles of 2 or 3 instructions, so it is a stretch calling it VLIW), most of the SIMD operations imaginable, and a super short 2-stage pipeline. And many cores: 36 in the first version, 72 in a later one. All of this clocked at a comfortably low 1.2 GHz, hence the short pipeline. Short pipelines are good. Trust me on that one.

The branch prediction was done just the right way: The programmer decided. Each branch was either going to branch or not and hardcoded in the instruction. If it was wrong, there was a 2-cycle mispredict penalty, thanks to the short pipeline. The compiler *GCC* usually made the right decision, but it could be overridden with a built-in call if not.

And they did the status register the right way: No status register at all. There were compare instructions that stored the result in a register as 0 or 1 explicitly. And the branch instructions branched on a register and a given condition. Simple. Just the way I like it.

All this cool stuff made programming it a breeze. Given that there were so many registers, the risk of the compiler getting confused and spilling registers to stack randomly was reduced significantly. Using intrinsics to exploit the SIMD stuff was simple. Other manufacturers have fumbled here; I will get back to that.

Anyway, a register was 64-bit, at last. Truly 64-bit. And they could be accessed as 64x1, 32x2, 16x4 or 8x8. And it did not stop there. It had the best instruction ever: *v1ddotpus*, which calculates two 4-way 8-bit dot products at the

same time and gives two 32-bit answers in 2 clock cycles. All instructions in the TILE-Gx were fully pipelined, so one could be started, and another completed each cycle.

And it had the instruction called *shufflebytes*, which shuffle bytes, obviously, from two sources into one. Think Intel's *mm_shuffle*, only useful. And there was *dblalign* which aligns two 64-bit sources across a 128-bit boundary. Nifty!

It did not have much floating-point support. What was there seemed tacked on: Each operation needed four instructions for a total of seven or eight cycles. And there was only multiply and addition/subtraction. There was double precision support too, but since there was nowhere to store the (unfortunately necessary) status bits, there were extra instructions for retrieving them. For floats, the bits were stored in the upper 32 bits of the register.

Unfortunately, this greatness did not come automatically. Just writing standard C code was not enough. Auto-vectorization had not taken off yet (or *drive-by optimization*, as I like to call it). I will get back to that later. But this CPU really flied when intrinsics were used. And it came with all the trace tools and simulators and stuff that is needed to write genuinely great code.

But the greatest invention in this CPU was the integrated mesh network. There were four of them, two available to the user. Each was 64-bit wide and clocked at the same 1.2 ghz. Sending data from one core to another was super quick. It had a switch on each core, so the number of cycles it took to transport 64 bits of data to another core was the number of steps along one axis plus the other. Other manufacturers at the time used increasingly complex ring bus solutions. I am not sure of the exact reason, but my guess is that Tilera had all the patents on mesh networks, and nobody wanted to pay them any royalties. But it is a guess. Anyway, some newer Intel CPUs have started using mesh networks. Good for them.

Unfortunately, like *Betamax*, quality does not always win. E*ZChip* bought Tilera in 2014 for a pittance: 130 million dollars in cash. And then *Mellanox* bought EZChip in 2016. And Nvidia bought Mellanox in 2019. Mellanox still had the TILE-Gx chips on their webpage before the last takeover, but there have not been any new developments since 2014. I had some inside info about a next generation architecture, but that is as dead as disco. Some hacker friends and I had petitioned for some custom instructions that would make video processing simpler. Guess it is not going to happen.

The TILE-Gx was fun, and some of the code I made for it are presented in other chapters. In the meantime, AMD released their underperforming architecture called *Bulldozer* in 2011. The idea was not that stupid: Instead of hyper-threading, it combined the computational resources of two cores, and both cores could pick what they needed from a larger pool. One core could blaze along with all units if needed. Great idea, except the decoder could not keep up, and it had only one floating point unit per two cores, making it useless for scientific applications. Or any floating-point application, really. D'oh.

Unfortunately, this also meant that development at Intel ground to a halt. The CPU world was stuck with four cores for a long time. The price per core for more than four cores was ridiculously high for several years. I did not see any large developments anywhere for a long time. A friend of mine kept his Intel *i7-2600K "Sandy Bridge"* system from 2011 alive for many, many years because of this. It felt like back in 2007, when Nokia had dominated the cell phone world since the start: They were the kings of the world, so there was no reason to invent anymore. That situation changed slightly when the iPhone launched later that year.

Luckily, in 2017, AMD came back from the dead. They had managed to make a completely new architecture, called *Zen*, built on a completely different concept (super-short version): Instead of a mega-die, they used multiple smaller dies that had maximum 8 cores and a large I/O die in the middle. Cheap to produce, and fast. And so many cores! I paid in blood for the current 8-core Intel CPU I have. Now, in 2020, the price for a 32-core AMD *ThreadRipper* CPU is about the same as the 8-core Intel one. And a good name is important! ThreadRipper sounds so much better than the lake-based names from Intel. When will they use the name *Crystal Lake*[2]? I really wonder what Intel is going to

[2] Probably never. Crystal Lake is known from the *Friday the 13th* movies. And *American Horror Story: 1984*, even though it's never mentioned. The fans know it's taking place there.

do. It will be exciting, or they will turn into the next *IBM*. Intel has always bounced back before, though, so I keep my hopes up for even more excitement in the CPU market in the coming years.

Looking forward there is one interesting architecture on the horizon: The *Mill*. It has been in development since 2003, and it presents a revolution in how memory is accessed, and how data is manipulated, and how branches are handled, the list goes on. Everything is improved upon. And there are no registers, only a belt where data falls off the edge when it is full, so the pipeline is open and exposed. It is an intriguing concept, and the main force behind it, Ivan Godard, has made several videos about the practical details on YouTube. Unfortunately, 17 years later, there are no chips on the market. I strongly advice to watch the videos. They really make you rethink a lot of stuff.

And yet, the basic concept is the same as before: It executes instructions one after another, and there are still no weird datatypes. Or lack of datatypes. And no garbage trucks driving around, collecting dynamically allocated garbage.

During all this time, the C language has prevailed. Today, standardization of the language is mostly sorted. The *C99* standard is useful and gives it a modern flair. But C is still the simple language it always has been. What is fed to the compiler normally translates to reasonable and readable assembler code. The most advanced abstraction is still the struct, which groups simple data units.

There catch is that C needs a programmer that knows what he or she is doing. It requires a keen interest in programming. It requires that you actually write code. At work and at home. If you do not have a computer with a keyboard at home, you are not a C programmer. You probably write some fluff in Phyreztorm and think it is programming. Guess what happens when Phyreztorm is replaced by *Phyrekloud* next year? You are obsolete. Unless, of course, you learn Phyrekloud. Which you will, since it is almost the same as the last one, only with slightly less horrendous bugs. And some new ones, which will be fixed in *Phyreball*. And if you think my naming scheme is childish, a sort-of language thing (I really do not know what it is) called *Node.js* from 2009 had a spinoff in 2018 called *Deno*. Really. When will *Done* hit the market? I am guessing 2027, after the glaring errors in Deno calls for another spinoff.

Back to the point: A good programmer will try out alternative solutions just for fun. I once had a bad manager who claimed much of programming was boring. I left that company shortly after. It turned out to be really boring there. The problem is that software managers eschew fun since it does not always produce results and never fit into their 5-year plans. I might be exaggerating slightly, since large corporations rarely plan further ahead than 3 months, but the point remains: How do they fit a box called "discover revolutionary new method to calculate vertical dot products on

our current architecture" into their PowerPoint plan for the next period? They have no clue what it means, and they slam the brakes on and demand more average code per hour instead. Getting them to understand that it is necessary to break new ground to make an innovative product is lost on them. I postulate

> *Corneliusen's two laws of software management (the first and the last): If all that's used is average, standard solutions, the product is guaranteed to be average.*

But let us get back to discussing C and its greatness: C allows you to write code that is dubious. There is an appropriate solution to that problem: Do not do it! A sign of a bad C programmer is that they have a superficial understanding of memory and pointers and allocation, and do not understand an important concept: Counting from 0 - zero. It is so simple, and so easy to fumble.

In May 2020 I read an article called *Memory Safety*[3] about, obviously, memory safety, in Google's Chromium project, used in Google's *Chrome* browser, and Microsoft's third attempt, or maybe it is their fourth. I have lost count. Anyway, quoting from the article:

> *Around 70% of our high severity security bugs are memory unsafety problems (that is, mistakes with C/C++ pointers). Half of those are use-after-free bugs.*

Things like that always happens when you do not have control. Maybe there are too many average programmers working on it? First, describing it as C/C++ is plainly wrong. It is either C, or it is C++. Not both. And C memory management is extremely simple. C++ can be done as complex as you like: Usually more complex than required; then the weird bugs start cropping up, like the aptly named use-after-free bugs.

In a section called *What we're trying* they state

> *In particular, we feel it may be necessary to ban raw pointers from C++.*

A craze that was popular some years back was to call programming a craft. I am not going to start a discussion about that but will try to explain the statement above in crafting terms. That might make sense to those who believe in it: Imagine it has been discovered that most head injuries on a building site is caused by bricks falling from the scaffolding. The obvious remedy would be to ban bricks. And, by default, all buildings would henceforth be built using wood. I am not sure the wooden high rises would be popular, or even safe. Next year, they will suggest removing all the saws, since they also cause so many injuries.

Removing the tools and building materials necessary to make a real building does not make a better building, just because there are some bad contractors. I would think replacing them would be better, but as I have already tried to explain, nowadays everybody is racing for the average by formalizing programming with stuff like "best practices" and "programming patterns": The bad programmers turn average, and the good ones too. They might figure it out, or they might not. Meanwhile, I am looking into other browsers. We had a good run, Chrome! I will give Microsoft *Edge* a try... oh wait. That ship has slid off the map.

So, then, my last point, and a very dominant problem in the programming world today: Recruitment. I am quite sure there are now more recruitment companies than actual programmers looking for work. The recruiters are clueless about programming, so they get a specification from a clueless software manager and follow it to the letter. They will only manage to hire good programmers by accident. In football terms, and it is a sport I care little about, they provide an endless supply of third division midfielders. The only thing that can be said about them is that they are completely average in every way, and they will never ever perform well out of their league. All they get is the career guys, who think they can learn programming like it is a craft and only do programming at work and become bad managers when they find out they cannot do it. The recruitment system of today fits nicely with the goal of everything being average.

[3] https://www.chromium.org/Home/chromium-security/memory-safety

And while we are at it, most such recruiters claim their customer pays competitive salaries. That is at best misleading, since everybody pays the same. I would call that average, but I guess it does not sound as good. I can document that, by personal experience: I have fended off a large set of such recruiters (around 40 so far, the number is increasing each week) by inquiring about the pay range, and the statistics are clear: 30% did not reply and moved on to easier targets, 25% tried to avoid the question, and the remaining 45% stated completely average ranges. It should be noted that, in Norway, determining the average salary is trivial: The two largest interest organizations for programmers (and others), *NITO* and *Tekna*, release excellent statistics released annually on their webpages. Somebody should inform the recruiters, often located in faraway countries.

Everything in software development is converging towards the average. Let us continue with something fun instead.