



## Advanced CPU Designs: Crash Course Computer Science #9

Crash Course: Computer Science

<https://youtube.com/watch?v=rtAIC5J1U40>

<https://nerdfighteria.info/v/rtAIC5J1U40>

Hi, I'm Carrie Anne and welcome to CrashCourse Computer Science!

As we've discussed throughout the series, computers have come a long way from mechanical devices capable of maybe one calculation per second, to CPUs running at kilohertz and megahertz speeds. The device you're watching this video on right now is almost certainly running at Gigahertz speeds - that's billions of instructions executed every second.

Which, trust me, is a lot of computation! In the early days of electronic computing, processors were typically made faster by improving the switching time of the transistors inside the chip - the ones that make up all the logic gates, ALUs and other stuff we've talked about over the past few episodes. But just making transistors faster and more efficient only went so far, so processor designers have developed various techniques to boost performance allowing not only simple instructions to run fast, but also performing much more sophisticated operations.

INTRO Last episode, we created a small program for our CPU that allowed us to divide two numbers. We did this by doing many subtractions in a row... so, for example, 16 divided by 4 could be broken down into the smaller problem of 16 minus 4, minus 4, minus 4, minus 4. When we hit zero, or a negative number, we knew that we were done.

But this approach gobbles up a lot of clock cycles, and isn't particularly efficient. So most computer processors today have divide as one of the instructions that the ALU can perform in hardware. Of course, this extra circuitry makes the ALU bigger and more complicated to design, but also more capable - a complexity-for-speed trade-off that has been made many times in computing history.

For instance, modern computer processors now have special circuits for things like graphics operations, decoding compressed video, and encrypting files - all of which are operations that would take many many many clock cycles to perform with standard operations. You may have even heard of processors with MMX, 3DNow!, or SSE. These are processors with additional, fancy circuits that allow them to execute additional, fancy instructions - for things like gaming and encryption.

These extensions to the instruction set have grown, and grown over time, and once people have written programs to take advantage of them, it's hard to remove them. So instruction sets tend to keep getting larger and larger keeping all the old opcodes around for backwards compatibility. The Intel 4004, the first truly integrated CPU, had 46 instructions - which was enough to build a fully functional computer.

But a modern computer processor has thousands of different instructions, which utilize all sorts of clever and complex internal circuitry. Now, high clock speeds and fancy instruction sets lead to another problem - getting data in and out of the CPU quickly enough. It's like having a powerful steam locomotive, but no way to shovel in coal fast enough.

In this case, the bottleneck is RAM. RAM is typically a memory module that lies outside the CPU. This means that data has to be transmitted to and from RAM along sets of data wires, called a bus.

This bus might only be a few centimeters long, and remember those electrical signals are traveling near the speed of light, but when you are operating at gigahertz speeds - that's billions of a second - even this small delay starts to become problematic. It also takes time for RAM itself to look up the address, retrieve the data, and configure itself for output. So a "load from RAM" instruction might

take dozens of clock cycles to complete, and during this time the processor is just sitting there idly waiting for the data.

One solution is to put a little piece of RAM right on the CPU -- called a cache. There isn't a lot of space on a processor's chip, so most caches are just kilobytes or maybe megabytes in size, where RAM is usually gigabytes. Having a cache speeds things up in a clever way.

When the CPU requests a memory location from RAM, the RAM can transmit not just one single value, but a whole block of data. This takes only a little bit more time than transmitting a single value, but it allows this data block to be saved into the cache. This tends to be really useful because computer data is often arranged and processed sequentially.

For example, let say the processor is totalling up daily sales for a restaurant. It starts by fetching the first transaction from RAM at memory location 100. The RAM, instead of sending back just that one value, sends a block of data, from memory location 100 through 200, which are then all copied into the cache.

Now, when the processor requests the next transaction to add to its running total, the value at address 101, the cache will say "Oh, I've already got that value right here, so I can give it to you right away!" And there's no need to go all the way to RAM. Because the cache is so close to the processor, it can typically provide the data in a single clock cycle -- no waiting required. This speeds things up tremendously over having to go back and forth to RAM every single time.

When data requested in RAM is already stored in the cache like this it's called a cache hit, and if the data requested isn't in the cache, so you have to go to RAM, it's called a cache miss. The cache can also be used like a scratch space, storing intermediate values when performing a longer, or more complicated calculation. Continuing our restaurant example, let's say the processor has finished totalling up all of the sales for the day, and wants to store the result in memory address 150.

Like before, instead of going back all the way to RAM to save that value, it can be stored in cached copy, which is faster to save to, and also faster to access later if more calculations are needed. But this introduces an interesting problem -- the cache's copy of the data is now different to the real version stored in RAM. This mismatch has to be recorded, so that at some point everything can get synced up.

For this purpose, the cache has a special flag for each block of memory it stores, called the dirty bit -- which might just be the best term computer scientists have ever invented. Most often this synchronization happens when the cache is full, but a new block of memory is being requested by the processor. Before the cache erases the old block to free up space, it checks its dirty bit, and if it's dirty, the old block of data is written back to RAM before loading in the new block.

Another trick to boost cpu performance is called instruction pipelining. Imagine you have to wash an entire hotel's worth of sheets, but you've only got one washing machine and one dryer. One option is to do it all sequentially: put a batch of sheets in the washer and wait 30 minutes for it to finish.

Then take the wet sheets out and put them in the dryer and wait another 30 minutes for that to finish. This allows you to do one batch of sheets every hour. Side note: if you have a dryer that can dry a load of laundry in 30 minutes, please tell me the brand and model in the comments, because I'm living with 90 minute dry times, minimum.



## Advanced CPU Designs: Crash Course Computer Science #9

Crash Course: Computer Science

<https://youtube.com/watch?v=rtAIC5J1U40>

<https://nerdfighteria.info/v/rtAIC5J1U40>

But, even with this magic clothes dryer, you can speed things up even more if you parallelize your operation. As before, you start off putting one batch of sheets in the washer. You wait 30 minutes for it to finish.

Then you take the wet sheets out and put them in the dryer. But this time, instead of just waiting 30 minutes for the dryer to finish, you simultaneously start another load in the washing machine. Now you've got both machines going at once.

Wait 30 minutes, and one batch is now done, one batch is half done, and another is ready to go in. This effectively doubles your throughput. Processor designs can apply the same idea.

In episode 7, our example processor performed the fetch-decode-execute cycle sequentially and in a continuous loop: Fetch-decode-execute, fetch-decode-execute, fetch-decode-execute, and so on. This meant our design required three clock cycles to execute one instruction. But each of these stages uses a different part of the CPU, meaning there is an opportunity to parallelize!

While one instruction is getting executed, the next instruction could be getting decoded, and the instruction beyond that fetched from memory. All of these separate processes can overlap so that all parts of the CPU are active at any given time. In this pipelined design, an instruction is executed every single clock cycle which triples the throughput.

But just like with caching this can lead to some tricky problems. A big hazard is a dependency in the instructions. For example, you might fetch something that the currently executing instruction is just about to modify, which means you'll end up with the old value in the pipeline.

To compensate for this, pipelined processors have to look ahead for data dependencies, and if necessary, stall their pipelines to avoid problems. High end processors, like those found in laptops and smartphones, go one step further and can dynamically reorder instructions with dependencies in order to minimize stalls and keep the pipeline moving, which is called out-of-order execution. As you might imagine, the circuits that figure this all out are incredibly complicated.

Nonetheless, pipelining is tremendously effective and almost all processors implement it today. Another big hazard are conditional jump instructions -- we talked about one example, a JUMP NEGATIVE, last episode. These instructions can change the execution flow of a program depending on a value.

A simple pipelined processor will perform a long stall when it sees a jump instruction, waiting for the value to be finalized. Only once the jump outcome is known, does the processor start refilling its pipeline. But, this can produce long delays, so high-end processors have some tricks to deal with this problem too.

Imagine an upcoming jump instruction as a fork in a road - a branch. Advanced CPUs guess which way they are going to go, and start filling their pipeline with instructions based off that guess -- a technique called speculative execution. When the jump instruction is finally resolved, if the CPU guessed correctly, then the pipeline is already full of the correct instructions and it can motor along without delay.

However, if the CPU guessed wrong, it has to discard all its speculative results and perform a pipeline flush - sort of like when you miss a turn and have to do a u-turn to get back on route, and stop your GPS's insistent shouting. To minimize the effects of these flushes, CPU manufacturers have developed sophisticated ways to guess which way branches will go, called branch prediction.

Instead of being a 50/50 guess, today's processors can often guess with over 90% accuracy!

In an ideal case, pipelining lets you complete one instruction every single clock cycle, but then superscalar processors came along which can execute more than one instruction per clock cycle. During the execute phase even in a pipelined design, whole areas of the processor might be totally idle. For example, while executing an instruction that fetches a value from memory, the ALU is just going to be sitting there, not doing a thing.

So why not fetch-and-decode several instructions at once, and whenever possible, execute instructions that require different parts of the CPU all at the same time!? But we can take this one step further and add duplicate circuitry for popular instructions. For example, many processors will have four, eight or more identical ALUs, so they can execute many mathematical instructions all in parallel!

Ok, the techniques we've discussed so far primarily optimize the execution throughput of a single stream of instructions, but another way to increase performance is to run several streams of instructions at once with multi-core processors. You might have heard of dual core or quad core processors. This means there are multiple independent processing units inside of a single CPU chip.

In many ways, this is very much like having multiple separate CPUs, but because they're tightly integrated, they can share some resources, like cache, allowing the cores to work together on shared computations. But, when more cores just isn't enough, you can build computers with multiple independent CPUs! High end computers, like the servers streaming this video from YouTube's data center, often need the extra horsepower to keep it silky smooth for the hundreds of people watching simultaneously.

Two- and four-processor configuration are the most common right now, but every now and again even that much processing power isn't enough. So we humans get extra ambitious and build ourselves a supercomputer! If you're looking to do some really monster calculations -- like simulating the formation of the universe - you'll need some pretty serious compute power.

A few extra processors in a desktop computer just isn't going to cut it. You're going to need a lot of processors. No.. no... even more than that.

A lot more! When this video was made, the world's fastest computer was located in The National Supercomputing Center in Wuxi, China. The Sunway TaihuLight contains a brain-melting 40,960 CPUs, each with 256 cores!

That's over ten million cores in total... and each one of those cores runs at 1.45 gigahertz. In total, this machine can process 93 Quadrillion -- that's 93 million-billions -- floating point math operations per second, known as FLOPS. And trust me, that's a lot of FLOPS!!

No word on whether it can run Crysis at max settings, but I suspect it might. So long story short, not only have computer processors gotten a lot faster over the years, but also a lot more sophisticated, employing all sorts of clever tricks to squeeze out more and more computation per clock cycle. Our job is to wield that incredible processing power to do cool and useful things.

That's the essence of programming, which we'll start discussing next episode. See you next week.