What We'll Do for Speed

Description: This week Steve and Leo examine the amazing evolution of microprocessor internals. They trace the development of the unbelievably complex technologies that have been developed over the past 25 years to wring every last possible cycle of performance from an innocent slice of silicon.

High quality (64 kbps) mp3 audio file URL: http://media.GRC.com/sn/SN-254.mp3
Quarter size (16 kbps) mp3 audio file URL: http://media.GRC.com/sn/sn-254-lq.mp3

Leo Laporte: This is Security Now! with Steve Gibson, Episode 254, recorded June 23, 2010: What We'll Do for Speed.

It's time for Security Now!, the show that covers your security, your privacy, and what you need to know to keep yourself safe on the interwebs. And here he is, the king of security, our very own Steve Gibson, man about town, man about GRC.

Steve Gibson: Normally people think that I'm over-caffeinated. But in this case, Leo...

Leo: I'm only on my second cup of coffee.

Steve: Nah, I'm just kidding.

Leo: How are you today?

Steve: Great. We have a great episode. I'm always excited when I'm able to bring something that I think is really going to be interesting to our listeners. One of the things that I constantly hear in our feedback is that people come away with something new that they didn't know every single podcast.

Leo: We like that.

Steve: Pretty much no matter what. And so it makes it worth their while, and it makes it

worth our while.

Leo: It certainly does.

Steve: Today I think we're going to - I've been projecting the completion of the series on the fundamentals of CPU technology...

Leo: Yeah. It's never done.

Steve: ...for quite a while. I think I'm finally running out. But before we switch to a number of things that we've got in backlog, and then once those are cleared out, plow into the fundamentals of networking, which is going to be our next big series...

Leo: Oh, yum.

Steve: ...I wanted to talk about, and that's what we're going to do today, what has happened over the course of the last 25 years in the internal design of microprocessors being pushed to unbelievable technology for the sake of speed. There's stuff in our micros which I think by the end of this podcast everyone is going to be thinking, I had no idea that's what they had done, that that's what was in there. It's just - it is truly remarkable what technology has been brought to bear that we've never touched on. When you and I first started talking about this as we fired up our connection, you were saying, well, you mean, like caching? It's like, oh, no, my friend. This is just unbelievable stuff. So...

Leo: Oh, I can't wait. It is a miracle, really. And it's such a commonplace miracle, as is often the case, that we take it for granted. And yet...

Steve: Well, exactly. Well, it's hidden.

Leo: Right.

Steve: And in fact, much of this is proprietary. And it's only from people scrutinizing patent documents and actually performing tests on the micros to see how they perform that what's been put into them has been reverse-engineered or has been gleaned from looking at patents that we go, aha ho, that's what this thing is doing. And it's just, well, remarkable. So, yeah, we've got a really great episode today.

Leo: Oh, I can't wait. All right, Steve. Do we have any security updates?

Steve: Well, we have - it's been a blessedly quiet week...

Leo: Yay.

Steve: ...after many weeks of a great deal of torturous, tumultuous news. I did want to mention something that I saw picked up in the news, which I had independently verified and dealt with myself, which was that this recent Mac OS X update, which we talked about last week, which was 313MB for me and various sizes depending upon what version and so forth, that it brought back the older, vulnerable version of Flash.

Leo: Oh, you're kidding. Now, they've done that before, and that's very infuriating. That's so bad.

Steve: Yeah. So it retrograded people who may have updated by putting - remember that we were at 10.0.45.2. And we went up to 10.1, which is now the official Adobe release. We had recommended that people jump ahead and use that 10.1 even when it was not yet official, when it was in prerelease, because it was known not to have the problem, which is now being very actively exploited on the Internet. There's lots of buzz about this big Flash problem. So...

Leo: Now, typically these exploits give the bad guy root access. But then usually the software he's using is not Mac software. It's Windows. So it's less, I mean, it's absolutely a threat to Mac users. But they're not prone to a lot of these online hacks right now.

Steve: Right. Well, the new model, of course, is a different threat model. It's this notion of as now is the term "weaponized email," which is sort of a version of spear phishing. And so we are beginning to see Mac exploits and Mac malware. I mean, it's beginning to happen. It's certainly lagging way behind where Windows is. And people with Windows are the larger target. But I just, sort of out of curiosity, I went through, under Safari - I also normally use Firefox on my Mac, but for some reason I was using Safari. Oh, I know, some reader had written that under HTTPS the lock had disappeared, and it was no longer possible to check your security certificate under Safari 5. And I said, oh, okay. So I fired up my Mac and fired up Safari 5. And it still is. It's a very tiny little lock in the far upper right corner which you have to click on, and then you can do everything. You can see what your security certificate is and so forth.

But while I was there I went through and looked at the add-ons that I had, the additional browser features. And I noted that, sure enough, Flash was back at 10.0.45.2. So I went to Adobe, downloaded it. Anyway, so just wanted to let Mac people know that they ought to check again to see what their version of Flash Player is. You can just go to Adobe.com, and right there is a little icon that says "Get Adobe Flash Player." Or you can go to get.adobe.com/flashplayer, which takes you immediately to the page where you download it and install it. And you do have to restart your browser in order for it to see the new version. And the good news is that Firefox's version of the Flash Player was updated at the same time. So just doing that for either one will take care of it for both.

Leo: Apparently some of the people in the chatroom are saying it didn't set theirs back if they'd already upgraded. So it may depend. But certainly worth checking to

make sure that you have the latest version.

Steve: Yeah, I'm sure I was updated. But, so, I guess...

Leo: Yeah, yeah, just something to be aware of.

Steve: Yup. Worth checking.

Leo: Don't be disappointed if it didn't downgrade you.

Steve: The only other real news is this - I just wanted to update our listeners because that's why we call ourselves "Security Now!," is what I guess I would refer to now as GoogleGate. This ongoing kerfuffle over Google's inadvertent collection of unencrypted wireless data. The most recent news is that Richard Blumenthal, who's the attorney general for the U.S. State of Connecticut, has now stated that attorneys general from 30 states have expressed an interest in joining them, that is, Connecticut, into an investigation into Google's collection of personal information over their unsecured WiFi collection. Which is continuing to be annoying.

And then what popped up in the news also this week is that the French data protection agency, it's called CNIL, their chairman, Alex Turk, has made the comment that in their early look at the data that Google turned over to them, which had been collected in France, he's quoted as saying that "data that are normally covered by ... banking and medical privacy rules" were found in the data. And IDG news also reported that CNIL had spotted passwords for email services and chunks from text messages.

And so my reaction is, yeah, I mean, we understand that's what's happening in unencrypted WiFi. Eric Schmidt, Google's CEO, he's saying, look, "No harm, no foul. Who was hurt?" Name a single person. And his point is that, yes, they recorded this on hard drives. They did it because the software that they use had defaulted in its default settings for doing so. But they never used the data. They never intended to. They didn't process it. And nobody was damaged by this.

Leo: I'm not a lawyer, but I know that intent is significant in criminal law, your intent.

Steve: Well, and suing for damages is - many of these random individuals who wanted to fire up a class-action lawsuit, the good news is you have to show damage.

Leo: Right.

Steve: And being annoyed is not damage.

Leo: So that's going to be the nub, is can you prove that you were damaged. Is it credible, given what we know, that Google - that this was an accident, or that Google didn't...

Steve: Yes.

Leo: It is.

Steve: It is entirely credible. They've shown the source code. They've turned that over. They've had it analyzed by a third party. I've looked at it myself and seen that the defaults that Kismet uses, which is a well-known open source WiFi collection tool, the defaults that it uses make sense. And they are record unencrypted payloads; don't bother to record encrypted payloads. As we know, encrypted WiFi payload is just pseudorandom noise. I mean, unless you go to huge extents to decrypt it, which Google wasn't doing. All Google wanted was the header information. As we understand, they wanted the MAC address and the SSID. That's all they wanted.

And then at the same time they were adding metadata, that is to say the current GPS coordinates and the signal strength, which Kismet does also add because that's one of the things that Kismet records in its own metadata. So they were just streaming all that stuff onto hard drives as they wandered around town, wandered around the globe actually, the whole world, sucking this stuff in. And, I mean, I don't have a single bit of doubt that this was inadvertent.

And I'm just wishing, what frustrates me is the wrong lesson is being learned here. I mean, people are all upset that Google recorded something that people were broadcasting. People have a responsibility for the fact that they're broadcasting this data. I mean, we understand this data is in the clear on this podcast, and that it's being broadcast. I read some interesting conversation in the security community with people saying, is it illegal for you overhearing your neighbors having a heated argument? No. I mean, they're shouting at each other out loud. You can't help but to hear it. Is it impolite? Well, maybe it's impolite to listen. But if it's being broadcast, as is a shouting match, then you're going to hear it.

I mean, and so, I mean, this is - what really frustrates me is unencrypted wireless is a massive problem. I mean, there's no bigger security issue today, I think. And the world could be learning an important lesson, which is unfortunately so far not being - it's not surfacing. What's surfacing is Google is bad for doing this, and that's ridiculous. So anyway.

Leo: Although we know, I mean, from case law we know that, for instance, if you are sitting out on a curb using somebody's unencrypted WiFi, just because it's unencrypted doesn't get you off the hook. People have been arrested for that, prosecuted for that, and even fined for that.

Steve: Which is entirely different than passively sniffing and not using.

Leo: Right. Yeah, I think, I mean, it's clear Google's going to turn back these lawsuits. But there is a public relations hit to this, and it mostly comes with people who aren't listening to this show. I'll try to do my best on the radio to talk about it. But I think that it's inevitable that, unfortunately, this state attorney general in Connecticut is doing Google a lot of damage and is really grandstanding, I think.

Steve: Yes, yes. Well, in fact, I had been meaning to ask you, Leo. I'd, if you're interested, like to come onto your...

Leo: Please do.

Steve: ...Saturday and Sunday show...

Leo: Please do.

Steve: ...because middle of next week Starbucks is going wireless and unencrypted.

Leo: Yeah. That's a bigger story. That's more important.

Steve: Yes. And so I thought that would - it would be good just to talk to all of the listeners of your Tech Guy show and say, look, yes, this is free. Yes, this is going to be nice, open WiFi. But understand the consequences.

Leo: Why don't we record that right after this show because I'm going to be at Foo Camp, and so we're recording the show ahead of time. In fact, this will be great. It'll give me another segment. You've got 12 minutes. You could do it twice, on Saturday and Sunday. So we'll record it right after this show because that is an important message. And we could mention this Google thing.

Steve: Oh, it would be perfect to mention it because it sort of ties into it because here's what - France is saying Google was recording people's email passwords. Well, they were.

Leo: Because people were sending them in the clear.

Steve: Exactly.

Leo: And I think we should probably also - I mention on the radio show all the time, but also mention this very simple thing, which is turning on WPA2 encryption is all you need to do. It's the one and only thing to do with a WiFi access point to secure it.

Steve: Yup. In fact, I did see a little blurb saying that the Wi-Fi Alliance, which is the formal standards body for WiFi, was going to be removing WEP encryption from the standard.

Leo: Good. Hallelujah.

Steve: Which as we know is, I mean, it's better than in the clear, but it's certainly not secure. There's technology, we've done podcasts about this that talk about in detail how it's now possible to crack the WEP key in about a minute.

Leo: Fantastic. That's really about time they dump that piece of junk.

Steve: Yeah. It was, as we know, it was an early standard that was designed with no consultation by cryptographers. And as the cryptographers began looking at it, the "security" of it just collapsed under scrutiny. So the lesson was learned. And WPA, the good technology, was designed correctly. So, and I think early on there was a problem with not - with in some cases using WPA because there were still devices and technology that was WEP only. But that's been years now. And, I mean, this is years old. And so I think it does make sense to retire it. The problem is, people are still just using no security. And I've used the term before, "the tyranny of the default."

Leo: Right.

Steve: I like the phrase because it says that most of the time people leave things in their default settings. Unfortunately, since the wireless access point and the wireless router people don't want a heavy tech support burden, they ship their access points and wireless routers defaulting to open, defaulting to no encryption. And so what happens is your typical user plugs it in, turns on their laptop, it finds it, and they go, wow, that was really easy. Uh-huh. Unfortunately it was too easy.

Leo: A little too easy.

Steve: Yeah.

Leo: I think that's changing. I know Linksys and others are starting to walk you through a secure process. Some of these companies are putting big buttons on their router that say "press this to be secure" and stuff like that.

Steve: Good.

Leo: So I think there's - of course they understand it's going to add to their tech support costs. They're going to get more calls. People are going to be confused. But I think that they realize they've got to - they can't just leave people out in the open

in the clear like that.

Steve: The good news is, in my own neighborhood, on some of my WiFi radios, I can see maybe 10 or 11 or 12 different WiFi nodes. Every single one of them now has a padlock.

Leo: Good.

Steve: And that was not the case a couple years ago.

Leo: No. Yeah, no, I remember going and doing a Netstumble - and I should try this again on my way to work - and recording hundreds of Linksyses. I mean, not even renamed. Let along WEP or WPA. I mean, they were called "Linksys." I'm sure the default password would work. So even if they turned on WPA I could just log in and turn it off. Crazy. Do you have any errata you'd like to...

Steve: I don't. I just have a short, very short and sweet little note from a listener of ours, John Levell, who's in the U.K. He said, "Steve, I'm a regular listener to Security Now!, so very familiar with the sort of feedback you receive for all your work, but just wanted to add some more. I just bought SpinRite. Five hours later my dead XP system is alive again. Many thanks for the quality of both your software and your podcast. J."

Leo: Isn't that nice.

Steve: So thank you, John, for sharing.

Leo: Isn't that nice.

Steve: He sent that from his iPhone.

Leo: Aw.

Steve: Yeah.

Leo: I feel the need, Mr. Gibson, for speed.

Steve: Well, so we've established sort of the original technology of computers, looking at the way, for example, early minicomputers like my favorite old PDP-8 operate, where memory is a bunch of words, and the words are broken into fields of bits, and the bits specify, for example, the opcode, the operation code, what the word of instruction is going to cause the hardware of the machine to do. And even then, even though the machine went from one instruction to the next, the execution of that instruction

internally took several phases.

You would fetch the instruction from main memory into the - we talked about the IR, the Instruction Register, where the machine would then look at the opcode to determine what this instruction was telling it to do. So there was a fetch of the instruction. Then there was a decode, where you'd decode what it is that you fetched.

Then comes to execute the instruction, whether it's incrementing the accumulator or adding a register to another, maybe jumping to somewhere else. And then in some cases you would be writing the results back, maybe writing the result of incrementing the accumulator back into the accumulator, or writing it back into main memory, if you were storing.

So from the programmer's view, the programmer sees this as atomic events, one instruction per word. The engineer who's designed the computer to do this sees that there's more going on. A single execution of an instruction actually requires many different phases - fetch, decode, execute, and then write back the results. So machines were being produced like that. And people naturally wanted them to go faster.

And what the engineers saw was that, well, you know, we fetched an instruction. Then we're decoding it, and we're executing it. But while we're doing those things we're not using main memory. That is, it's waiting for the next fetch. And so the concept dawned on them, and this actually happened on the mainframe level in the late '60s, this notion of sort of overlapping things. And the best example, sort of I think the model that's clearest is, because we've all seen examples of it, is the automobile assembly line - which, as I understand it, Ford invented to create his cars, the idea being...

Leo: Just a side note, by the way, we're going to be going to visit the Ford assembly line on July 30th.

Steve: Who, "we"?

Leo: Me. Who, "me."

Steve: Oh, cool.

Leo: Yeah, and I'm going to bring the live camera, and we're going to actually show the state of the art in modern assembly, which I can't wait, their Dearborn plant.

Steve: I would love to see that because you only get little snapshot snippets of...

Leo: I know.

Steve: ...pictures with robot arms swinging stuff around in the air.

Leo: I know, I'm so excited.

Steve: That will be really cool.

Leo: I'm going to go see where my Mustang was born. Anyway, sorry, didn't mean to interrupt, go ahead.

Steve: So the idea with an assembly line is that, at every stage of assembly, you do a little bit of work towards producing a finished car. The time required to produce one car is the time it takes to go the length of the assembly line. But once the assembly line is full of partial cars being assembled, the rate at which cars come out is much faster than the total time it takes for a car to move through the assembly line. So...

Leo: Wait, now, let me think about that. The cars come out faster than it takes for a car to go through the assembly line.

Steve: Yes. So say that you had an assembly line of 10 stages.

Leo: Yeah.

Steve: And that each stage took a minute.

Leo: Okay.

Steve: Well, when you start making a car...

Leo: It takes 10 minutes.

Steve: It's going to take 10 minutes for that car to go all the way through the assembly line.

Leo: Oh, but then cars will come out one every minute.

Steve: Exactly.

Leo: Cool.

Steve: Once the assembly line is full, then they come out every single minute.

Leo: Got it, okay. I'm glad - sorry. I'm stupid, but I needed to understand that. Okay.

Steve: And so in processor technology we call this a "pipeline." And virtually every machine now being made, and actually made for the last two decades, has been "pipelined" to one degree or another. So let's first apply that to the very simple model of this machine which fetches the codes, executes, and writes back. The idea with a pipeline there would be that you fetch an instruction, then you start decoding it. Well, while you're doing that, memory is free. You're not using memory. So most instructions, most code is sequential. That is, we know that after normal instructions are executed, the program counter is incremented by one for the next word, which is then fetched. And the one after that and so forth.

That changes in the case of jump instructions, which jump us to somewhere else; or branch instructions, which may or may not branch to somewhere else. But in general it's a safe bet that we're going to be moving forward. So the engineers who wanted more performance out of the system basically - and this will be a recurring theme through this podcast. You look at the various components of your system and think, how can we keep them busy all the time? How do we get the most performance out of it? Well, it's by keeping all the pieces busy.

So if, while we're decoding an instruction we just fetched, we assume that we're going to be executing the next one here in a while, well, go ahead and fetch it. Get it read from memory. And similarly, after that first instruction's been decoded, then it's time to execute it. Well, meanwhile, at that point the decoder is not busy because it just did its work on the first instruction. Well, now we've got the second instruction that we fetched while the first one was being decoded. It can now be decoded.

And so the analogy is exactly similar to the assembly line where instructions move through several stages. And once they get going, rather than an instruction having to go all the way through before you even start on the next one, you're able to make some assumptions that allow you to basically create an assembly line for computer instructions, just like you do for cars.

Now, it gets, from that simple sort of start, then things really get interesting because one of the things that happens is that instructions may interact with each other. That is to say, if we were to add two registers - say that we had a machine with multiple registers, as all contemporary machines have now. Back, you know, that PDP-8 had just the one accumulator, which you sort of ended up using as a scratchpad. Now we've got 8, 16, 32, lots of registers. So say that an instruction that you read was adding two registers together, that is, adding one into another, and that the instruction that followed took the value from that add and stored it. Well, so now we have a problem because we have instructions in this pipeline which interact with each other.

So over time engineers have made these pipelines longer because they'd like to get as much simultaneity going as possible. But they've also had to deal with the fact that there can be interactions, and often are, between instructions that are in the pipeline at the same time. So the first thing that's done is that instructions are broken into smaller pieces. They're called "micro ops" (uOps).

So, for example, say that we had a simple instruction. We've talked about how the stack works; how, for example, when you push something on the stack, what happens is the stack pointer is decremented to point to a new, lower location in memory. And then the

value that you're pushing is written to the location where the stack pointer is pointing, sort of in this scratch pad. So that operation, a single instruction, "push a register," can actually be broken into two micro operations. The first one is decrement the stack pointer. The second one is copy the register to where the stack pointer is pointing.

And imagine another instruction, like adding a register to what's in memory. Well, to do that you have to read out what's in memory. Then you have to add the register to what you read out and then write that sum back to that same location in memory. Well, that's three micro operations. So what the processors do now is they take these sort of what look - the programmer sees them as instructions, but they're actually complicated enough that they require - they can be broken down into smaller pieces. So the processor fragments these single instructions into multiple micro operations and then basically pours them into this pipeline, which is getting increasingly long, in some cases as long as, like, 20 stages of, like, staging of instructions.

Now, one of the things that engineers noticed was that some instructions, like this - imagine the instruction I talked about where we're wanting to add a value to something in memory, where we're having to read the thing out of memory, then sort of into some internal temporary location that isn't even named. Then we add a register to that and then write it back out. Well, so we've taken that single instruction and broken it into these three micro operations.

Now imagine that there's an instruction behind it, that is, it actually is later in the code, that's doing something else entirely. It's adding two registers that aren't related to these micro operations. What the engineers realized was, while the computer was out fetching the value to be added to, it had already fetched more instructions behind. And the ones it had behind were independent of the outcome of the instructions that it was working on currently. And, for example, while fetching something from memory, the machine's adder, the ALU, the Arithmetic Logical Unit, was idle.

So the processors of today are able to realize that they've got other things they can be doing which are independent of the outcome of earlier instructions. So why not execute them out of order? That is, literally rearrange these micro operations in the pipeline so that things that are taking longer can literally be passed by instructions which can take advantage of resources in the chip which are not currently in use.

And so what we've ended up with is this amazing technology which pours instructions in the front end of the pipeline, fractures them into smaller sort of individual granules, which need to be executed in order for that to happen. Then logic which is sophisticated enough to look at the interdependencies between these micro operations and reorder them on the fly so that the assets that the chip has, like Arithmetic Logical Units, like a floating point processor, like instruction decoders, all these assets are being maximally used.

And in fact one of the things that happened was that processers went so-called "superscalar." What I've described so far is a scalar processor. A superscalar one is one which is actually able to execute more than one instruction per cycle. That is, normally you would be retiring instructions when you're done out of this pipeline, sort of at a given speed.

Well, if you have enough assets to execute instructions, there's no reason you can't go faster than a hundred percent. And so superscalar processors go faster than a hundred percent. They've got, for example, there are some that have four ALUs and two Floating Point Units. And so they're literally able to be doing four additions at once. Sometimes those are part of a very complex instruction, or sometimes they're part of different

instructions.

The point is, the processor has become smart enough to break all of these down into little subfunctions and then sort through them, analyzing the interdependencies among these subfunctions and taking advantage of anything that might require a delay in order to say, oh, wait a minute, we've got a guy back further here who isn't dependent upon any of our outcomes. And we've got a free adder. Let's do that addition right now. And if you think for a minute about the logical complexity of any instructions which you might encounter, and having to, on the fly, I mean, we're talking - there's no time to do this, either. This is not slowing things down. The goal is to speed everything up.

So there's no - you can't even catch your breath. This is all happening billions of times a second. At gigahertz speeds this is all being managed. So now we have a system which is able to do, literally, sucking instructions in, cracking them down, rearranging them on the fly, looking at interdependencies. Well, that wasn't enough for the engineers. Management said "faster, faster, faster." And so the engineers are like, wait a minute. We're going as fast as we can.

Well, what they realized was that wasn't true because there was still a way they could get a little more clever. I used the word "retiring an instruction" before. And that's a term used in this art where you finally say - you, like, write the results of the instruction back out. So inside this pipeline you've got an amazing amount of resources. You've got unnamed registers. By that I mean they're not like the register 0, register 1, register 2, or AX, BX, CX. That is, they're not the registers that the programmer sees. These are like temporary scratchpad registers which are not visible to the outside world, not visible to the instruction stream. But they're used, for example, if you were adding something to memory where you've got to read that into somewhere in order to add something to it. So that's an unnamed register.

So when you retire an instruction, you're sort of writing its results back out to, like, the real registers, to the programmer-accessible registers. But the engineers realized that in some cases they did have a result which a later instruction was waiting for, even though they hadn't yet retired the earlier instruction out to, for example, writing to, like, the AX register. They did have it in the pipeline.

So they added a whole 'nother layer of nightmare by allowing results to be forwarded, and that's the term that's used, within the pipeline to, like, track this so that partially executed instructions which had not yet been retired could have their results sent sort of back into the future in order to allow instructions that had stalled because they were dependent upon an outcome which hadn't been resolved yet. And all of that exists also. So what we have now is something unbelievably complicated.

Now, what happens if you hit a branch? Because branching, any change of linear flow is the worst possible thing that can happen. Think about it. We've got all this happening. We've got 20 instructions maybe that have been taken apart, all under the assumption, remember we made one fundamental assumption at the beginning, which was we're going to go linear. All of this sucking in things ahead of where we are assumes we're going to use them. All of this work says that we know where we're going.

Except when we come to a conditional branch, or even a jump that's going to go somewhere, suddenly everything changes. We now don't know whether we're going to keep going or go somewhere else until later in that instruction's phasing. Remember, now instructions are being cracked apart. They're being decoded. They're being executed. There's, like, all this work being done before the outcome of the instruction is known.

The problem is, if it's a branch instruction that might change the sequence, if it does change, if it's branching us to somewhere else, well, everything behind that instruction has to be scrapped. So the entire pipeline has to be dumped. And we stall until we are able to then load a series of instructions from the new location and sort of get all this going again.

Leo: And that's what screwed up Prescott because I think their prediction wasn't good, or their pipelines were too long, and they got a lot of dumps.

Steve: Well, yes. So having developed this amazing complexity for dealing with, I mean, like, just incredible acceleration of performance, as long as you go straight, the second you change away from that, that linear execution, you're in serious trouble. So engineers realized that branch prediction was crucial, that is, literally being able to guess without knowing what a branch was going to do.

Well, the way they've come up with doing this, there was a first level. You can imagine a simple-minded way which says, okay, let's assume that the branch that we encounter, if we've ever encountered it before, is going to do the same thing. So that sort of makes an assumption that branches generally do whatever they're going to do. In fact, microprocessor designers realized that many branches that are branching backwards are at the bottom of a loop, sort of a loop of code which tends to get executed a lot, and then finally isn't executed. So the branch, a branch backwards tends to be taken, as opposed to a branch forward. So there was some simple-minded sort of branch guessing that way.

Then they said, well, wait a minute. Let's record the last outcome of the branch and assume that it's going to do the same thing again. So an early branch predictor simply did that. And the idea was that you would take a chunk of the lower address bits, so like the least significant address bits in the instruction counter; and you would, for every one of those address bits, you'd create a table that had a single bit in it, which recorded a branch at this location did the following thing last time. It was taken or it wasn't taken.

Now, we're not talking about having a bit for every branch in the computer. We're saying that we're going to have sort of a bit, maybe 256 bits, for, like, the lowest byte of the instruction. So branches could collide with each other. A branch that was exactly 256 words further down would end up having the same least significant byte of address. So its bits would collide with each other. There's nothing we can do about that. But the probability of that is relatively low. And so there was always this cost versus performance tradeoff that's being made.

But the engineers weren't happy with just using a single bit because imagine that you had a branch which, in the code, literally alternated what it did every other time. It turns out that's also very common. Well, that would literally mean that every prediction you would make was wrong. If you remembered what it did last time, and you assumed it was going to do it again, and the logic in this branch was in every other logic, then you'd always be guessing wrong. And so the performance would just be abysmal. You'd get no benefit from your pipeline. You'd be constantly dumping the pipeline and then needing to refill it.

So the developers came up with a two-bit branch predictor, which they call a saturatable or a saturating counter, the idea being that - so two bits could obviously have four states. You could be zero zero. And then if you count up, you go to zero one. You count again, you go to one zero. And again, you go to one one. So those are the possible values. So the idea of a two-bit branch predictor was that, if you took the branch, you

would increment this two-bit counter, but never more than one one. So that's the saturating part. It would saturate, it would go to one one and then just stay there. If you did not take the branch, you would decrement the counter down to zero zero, but then you never go below zero zero. It saturates at the bottom end also.

So what this gave you was a better, sort of more of a probability. You could, if you generally took the branch, but not always, this counter would - it would still make a mistake, but it wouldn't change its mind completely. So if you generally took a branch, even if you occasionally didn't, it would still remember that you generally took it. So it would, again, it would generally be guessing correctly. And so that increased the performance of branch prediction substantially. But there was still a problem, which was that there were patterns of branching which this simple-minded two-bit predictor couldn't handle.

And so in real-world applications it was better than nothing, way better than nothing. But some other engineers realized, hey, we can do something even more clever. We can do a two-level prediction. So what they created was a shift register of bits which was whether the branch was taken or not, in history. And it wouldn't be very long. Maybe let's say for a discussion that it's only four bits long. So the shift register is remembering whether branches actually were taken or not. And every time we come to a branch, we first of all look at the least significant byte of the address to choose one of 256 whole worlds.

So each possible location in memory, with this 256 cycle, has its own entire little branch prediction world. Okay, so within that world is a four-bit shift register that remembers for that branch, or branches at that location in memory, whether the branch was taken or not. Okay, those four bits, we know that four bits gives us 16 possibilities. Those four bits are used to choose one of 16 of our little two-bit saturating counter predictors.

And what we end up with is literally pattern recognition, where over time this acquires a knowledge of any sequence of up to four long of branches and not branches being taken. That will be recorded in the two-bit predictor which will tell the computer with very good probability whether the branch will be taken again or not. And these predictors have grown in length and in size. And so remember that there's one of these whole predictors for each of a number of different locations in memory where these branches could fall.

So now what we've done is we've got this pipeline sucking in instructions, cracking them down, looking at their interdependencies, reorganizing them on the fly, taking it - we've decoupled the Arithmetic Logical Units and the floating point processors and the instruction decoding and all of this so that those are all now separate resources which are being assigned and used as soon as they can. As soon as we're able to see that we know enough to allow one of these micro operations to proceed, we do.

At the same time, the system is filling up the pipeline at the top using the results, assuming we're going linearly, unless we hit a branch or a jump, and then recording the history and literally learning the pattern of the past sequence of branches in the code and sort of heuristically developing an awareness of pattern recognition of whether - I mean, so that it's able to guess with as much as, it turns out, 93 percent probability whether a given branch will be taken or not, only missing about 7 percent of the time. And when it's wrong...

Leo: Is that the average for all processors, or...

Steve: Yes, state-of-the-art prediction now.

Leo: That's amazing.

Steve: I know. It's just incredible.

Leo: Just amazing. It's like that old joke, how do it know? It's like predicting the future, really.

Steve: It's like 6.75 percent misprediction, so about 7 percent misprediction. 93 percent of the time they're able to guess right.

Leo: Wow.

Steve: And so making a mistake is expensive in prediction because we have to flush all the work we were doing, and then go somewhere else. But 93 percent of the time we're able to get it right.

Leo: Somebody's asking in the chatroom, this isn't security. Well, in a way it is. This is a series Steve's been doing all along on the basics, the fundamentals of computing. In fact, from day one on Security Now! you've really done a great job, I think, of getting people up to speed with these fundamentals, things you have to understand to understand security; right? These are not completely incidental to security.

Steve: It certainly is the case that everything is interrelated. For example, I'm thinking as I'm working toward getting going on CryptoLink, the VPN product that I'm going to do next, well, encryption performance and decryption performance is very important. And understanding the internals of what the chip is doing really does allow a developer who wants to truly optimize their code to arrange the instructions so that the logic in the chips have the most latitude for working. And certainly performance has been something that we've been questing after forever.

Leo: Yeah. And we're getting it with this amazing pipelining and parallelism and so forth.

Steve: So the engineers have got this incredible pipeline built. They've got now this amazing branch prediction system. And then they realize that they've still got a problem because they suck in a return instruction into the top of the pipeline. Well, we know from having discussed this before what a subroutine return does. When we call a subroutine, we're using the stack. We decrement the stack pointer and put the address of the instruction after the call on the stack so that later, once that subroutine has finished, it's able to do a return instruction which looks on the stack for where to return to, which has a beautiful effect from a programmer's standpoint of bringing us right back to the instruction following the one which invoked a subroutine.

Well, one of the first things a subroutine does, because most subroutines don't want to

mess up what was going on when they were called, they'll push a bunch of registers value onto the stack themselves so that they can be popped off the stack and restored prior to returning. That allows them to have sort of those registers to work with and then not mess up what was going on in the main code that called the subroutine. Okay, so with that in mind, visualize what's going on in the processor now with the pipeline, where the pipeline is full of instructions toward the end of the subroutine, and then the subroutine is finished, and it does a return.

Now, the problem is that the return uses the value on the stack. But the thing that the subroutine is doing just before it returns is cleaning up its registers, getting their values off the stack in order to restore them to what they were. And this is happening further down in the pipeline. Which means the stack pointer is going to be changing a lot, and there's no way we can use, there's no way we can execute any of the return instruction until literally we get - we know what the stack pointer's going to be. And then we have to go read where it's pointing, get that value. That tells us where to return to.

So then we start fetching instructions from there. Which means a return instruction is deadly. It literally brings everything to a halt because we don't - we don't know where the stack pointer will be because the instructions typically occurring, all of those instructions just before the return are changing the stack pointer as they pop the values of the registers back off the stack into the registers so that they're restored when we go home.

So the engineers scratch their head for a while, and they say, wait a minute. What we need is an internal call stack. We need our own private stack because we know that, more often than not, subroutines nest perfectly. That is, some code calls a subroutine, which will return. Or maybe that subroutine calls a subroutine, but that one returns to the one that called it, and then it returns to the one that called it. In other words, there's a nesting which is almost always followed. Which means that this incredible execution unit in the processor now maintains its own call stack. When it sees that it has been jumped to a subroutine, it records internally the address of the instruction after that call on its own little stack. There's no instructions to get to it. Programmers can't see it. It's completely invisible.

The call stack ranges from 4 to 32 entries in modern processors now. And so what happens is, since the internal pipelining miracle has recorded this, the second a return instruction is seen, which is just a byte, for example, in an Intel instruction is just a 60 hex, a six zero hex, the second that byte is seen, the system says, ah, that's a return instruction. We don't have to wait for anything. We can immediately pull where we know it's ultimately going to return to off of our own internal stack and continue without interruption sucking in more data from the point we're going to be returning to, without missing a beat. So that's another level of what was added.

Now, once all of this was finished, and this was maybe, oh, about a decade ago we had this level of technology, there was still some unhappiness with the contention for resources. That is, there was still not what's called "instruction-level parallelism." There still was, like, the ALUs and the Floating Point Units. They were sitting around not being used all the time. The engineers weren't able to get them busier because there was still too much interdependence among these micro operations that they were just - they couldn't get enough, they weren't able to use the resources fully.

Well, this is when this notion of simultaneous thread execution occurred to them, which Intel calls "hyper-threading." I mentioned it a couple weeks ago in passing. I couldn't do it justice because we didn't have enough foundation to understand what hyper-threading is. Well, we do now, after the last 45 minutes of this. What hyper-threading is, is the

recognition that there is what's called "register pressure." There is not enough freedom of value assignment among registers. There's just too much interdependency. But if we had a whole second set of registers, if we duplicated everything, then where some microinstructions are fighting with each other, too interdependent, where they're having to wait for results to finish before the later ones can start, therefore assets like the Arithmetic Logic Unit and Floating Point Unit are sitting around being unused, if we have another physical thread, that is, we have another whole set of registers, well, there, because it's a different thread, they're logically disconnected from the first thread's registers. There is no conflict at all possible between these separate banks of registers.

So what hyper-threading does is, I mean, and this - talk about it being confusing already. This literally pours instructions from two entirely different threads of execution down into the same pipeline, breaking them all up, keeping them all straight, realizing that these micro ops and these registers are actually different from those micro ops and those registers. So now we have - we've doubled the opportunity for exploiting these fixed assets, the Arithmetic Logical Unit and the Floating Point Unit, being able to keep them busy much more of the time, which is what hyper-threading does. Essentially, it doesn't duplicate the entire system, but it allows us to pour two different threads of execution into the same pipeline and get a tremendous boost, I mean, it's not like doubling. We don't get double because the resources weren't that underutilized. Typically it's about a 25 percent gain, which in this quest for performance is better than a kick in the head.

So, lastly, with all of this, sort of with this much industry having been expended in order to satisfy essentially CISC, that is, Complex Instruction Set Computers, the guys designing the RISCs were just dizzy. They said, okay, wow. Do we want to do the same thing? Are we going to basically duplicate this insanity? RISC architecture is different in a number of ways. Fundamentally, the RISC concept was designed to prevent there from being a lot of this kind of like available performance boost because the instruction design just doesn't get itself into trouble so much.

One of the very clever things that RISC instructions do is there's something called "conditional instructions" and something called an "explicit condition code update." Now, what that means is that, notice that we have a stream of instructions that are being executed by the processor, and then a branch instruction is often skipping over some. There'll be something that, like, you don't want to execute in this certain case. So you jump ahead 10 instructions or five instructions or something. You're skipping over them. Which is many times what a branch will do.

What the RISC designers realized was, at the expense of some more bits in the instruction word, and it does widen the instruction word a bit, they could make what's called "conditional instructions" instead of branches, that there are still branches and jumps, and those are being optimized still very much the way they are in CISC instructions, with branch prediction and so forth. There's no way around that. But essentially the RISC guys said, wait a minute. If we just want to skip over five or six instructions, for example, if the result of an add was zero, or if the result did not overflow and the carry bit was set, why not add to any instruction some additional bits that say "execute this unless the condition code is zero." Which means that we've saved ourselves a branch. We don't have to branch over those instructions. We can make the instructions themselves just sort of skip over themselves. The instruction says "only execute me in this certain case," that is, the case where we wouldn't have taken the branch.

So what this did was, this allowed a very aggressive forward-fetching pipeline to go in a straight line. And we understand why pipelines like to go in a straight line. We were talking about that before. This allows the pipeline to fetch ahead. And even though it may not be executing instructions, it saves all of the possibility of a branch misprediction

because we don't have a branch at all.

Now, the other trick is, if you had a group of instructions which you collectively wanted to execute or not in a certain case, if you were executing them, you wouldn't want them to change, like, the state of the carry bit or the zero bit or any of the condition codes because then that would mess up the conditional execution of the instructions that followed. So the other thing that was added, in addition to this notion of a conditional, conditionally executed instruction, is the ability for the instruction not to modify the condition code when normally it would. You might be, like, doing some addition. And normally the add instruction will set a flag saying, oh, the outcome was zero, the outcome sent the carry bit, the outcome was not zero, you know, various condition code situations like that.

So what was done was a bit was added that said, do the add, but don't change the condition code because we're wanting to continue the instructions afterwards along the same - to have the same effect as the one we just executed based on a condition code which was set deliberately earlier in the path. And so that was essentially the final optimization that the RISC guys brought into the design of the instruction set, which further made pipelines able to absorb this huge number of instructions, sort through everything, and perform really this just overwhelming job of making processors incredibly fast.

Leo: It is such an amazing, mind-boggling thing, especially when you think that we're operating now at the microscopic - microscopic - at the level of a molecule's width, in some of these newer 45-nanometer processors. It's truly amazing.

Steve: Well, yeah, and I would imagine that probably everyone listening to the podcast has at one time or another seen one of those very cool photomicrographs just of a processor chip, sort of as if it was taken - it looks like a satellite photo of a city. And you look at that, and you think, my goodness, look at that, I mean, you can just tell by the level of detail in there that an incredible amount of something is going on. Well, what we've just described is what that something is. This technology is what has increased the power consumption, increased the size, increased the cost, but dramatically allowed the performance of these processors to increase.

And this, what we described today, this kind of incredible, out-of-order execution, branch prediction, internal call stack, register renaming, all of this is in all of today's processors. It's just being taken for granted now. It's the way we have the kind of performance that we do. Without any of this stuff, we'd be back with an 8088 running at 4.77 MHz.

Leo: There was a really good book, must be 10 years ago now, called "The Silicon Zoo," where they had those little pictures, the pictures of the stuff. Of course, it's so old now, it's changed a lot. But these photomicrographs, if you search for Silicon Zoo, they're still online. Some pretty amazing pictures. You can tell how old this site is, though, because it says "This is going to take a minute at 28.8." It's a big image. But you can do a little googling, and you'll find it. Fascinating stuff, Steve. Once again, you've done a great job of explaining how this stuff works. And networking next. But next week we're going to do a Q&A, I think; yes?

Steve: Yup. We have a Q&A, and then I'm going to - many listeners have said, hey, Steve, I thought you were going to tell us about LastPass. We're using it. We want to

know we should be and it's safe. So that's queued up for two weeks from now. We'll do a Q&A next week, and then I'm going to explain in detail the cryptographic protocols for LastPass and how the whole system works.

Leo: Oh, great. That's great. If you have Q&A questions, GRC.com/feedback is the place to go. He's got a feedback form there. GRC is a great place to go for not only SpinRite, the world's best hard drive maintenance and recovery utility, but also this show. 16KB versions are there for bandwidth-impaired fellas and gals. Of course I love the transcripts, it's a great way to follow along. And I suspect more than one teacher is using your lectures on computer fundamentals in their classes. So I think transcripts would be very helpful in that case, as well. You're more than welcome to do that.

I hope that you understand you don't have to get our permission to use these podcasts. They're Creative Commons licensed. Attribution-Noncommercial-Share Alike is the license. You can find out more at TWiT.tv at the bottom of the page there about our license. And you're more than welcome to use these. In fact, I think it's great if you do in courseware. Somebody was asking in the chatroom.

Steve is also the author of many great freebies which you'll find at GRC.com. And he's got a Twitter account now. Be careful. He's got more than one. He's got several. In fact, he's got the main account, which is @SGgrc. He's got the account - are you still posting articles about tablets?

Steve: Haven't for a while, but when something comes up I will definitely do that.

Leo: That's @SGpad. And then the corporate account, @GibsonResearch. That's all on Twitter. And his new blog, steve.grc.com. Did I get that right?

Steve: You did.

Leo: All right, my friend. God, I love this stuff. Fascinating stuff. Thank you so much.

Steve: Talk to you next week, Leo.

Leo: Take care.