CHARLES LEISERSON: So today, we're going to talk about assembly language and computer architecture. It's interesting these days, most software courses don't bother to talk about these things. And the reason is because as much as possible people have been insulated in writing their software from performance considerations. But if you want to write fast code, you have to know what is going on underneath so you can exploit the strengths of the architecture. And the interface, the best interface, that we have to that is the assembly language. So that's what we're going to talk about today.

So when you take a particular piece of code like fib here, to compile it you run it through Clang, as I'm sure you're familiar at this point. And what it produces is a binary machine language that the computer is hardware programmed to interpret and execute. It looks at the bits as instructions as opposed to as data. And it executes them. And that's what we see when we execute.

This process is not one step. It's actually there are four stages to compilation; preprocessing, compiling-- sorry, for the redundancy, that's sort of a bad name conflict, but that's what they call it-- assembling and linking. So I want to take us through those stages.

So the first thing that goes through is you go through a preprocess stage. And you can invoke that with Clang manually. So you can say, for example, if you do clang minus e, that will run the preprocessor and nothing else. And you can take a look at the output there and look to see how all your macros got expanded and such before the compilation actually goes through.

Then you compile it. And that produces assembly code. So assembly is a mnemonic structure of the machine code that makes it more human readable than the machine code itself would be. And once again, you can produce the assembly yourself with clang minus s.

And then finally, penultimately maybe, you can assemble that assembly language code to produce an object file. And since we like to have separate compilations, you don't have to compile everything as one big monolithic hunk. Then there's typically a linking stage to

produce the final executable. And for that we are using ld for the most part. We're actually using the gold linker, but ld is the command that calls it.

So let's go through each of those steps and see what's going on. So first, the preprocessing is really straightforward. So I'm not going to do that. That's just a textual substitution.

The next stage is the source code to assembly code. So when we do clang minus s, we get this symbolic representation. And it looks something like this, where we have some labels on the side. And we have some operations when they have some directives. And then we have a lot of gibberish, which won't seem like so much gibberish after you've played with it a little bit. But to begin with looks kind of like gibberish.

From there, we assemble that assembly code and that produces the binary. And once again, you can invoke it just by running Clang. Clang will recognize that it doesn't have a C file or a C++ file. It says, oh, goodness, I've got an assembly language file. And it will produce the binary.

Now, the other thing that turns out to be the case is because assembly in machine code, they're really very similar in structure. Just things like the op codes, which are the things that are here in blue or purple, whatever that color is, like these guys, those correspond to specific bit patterns over here in the machine code. And these are the addresses and the registers that we're operating on, the operands. Those correspond to other to other bit codes over there. And there's very much a-- it's not exactly one to one, but it's pretty close one to one compared to if you had C and you look at the binary, it's like way, way different.

So one of the things that turns out you can do is if you have the machine code, and especially if the machine code that was produced with so-called debug symbols-- that is it was compiled with dash g-- you can use this program called objdump, which will produce a disassembly of the machine code. So it will tell you, OK, here's what the mnemonic, more human readable code is, the assembly code, from the binary.

And that's really useful, especially if you're trying to do things-- well, let's see why do we bother looking at the assembly? So why would you want to look at the assembly of your program? Does anybody have some ideas? Yeah.

AUDIENCE: [INAUDIBLE] made or not.

CHARLES LEISERSON: Yeah, you can see whether certain optimizations are made or not. Other reasons? Everybody is going to say that one. OK.

Another one is-- well, let's see, so here's some reasons. The assembly reveals what the compiler did and did not do, because you can see exactly what the assembly is that is going to be executed as machine code.

The second reason, which turns out to happen more often you would think, is that, hey, guess what, compiler is a piece of software. It has bugs. So your code isn't operating correctly. Oh, goodness, what's going on? Maybe the compiler made an error. And we have certainly found that, especially when you start using some of the less frequently used features of a compiler. You may discover, oh, it's actually not that well broken in.

And it mentions here you may only have an effect when compiling at -03, but if you compile at -00, -01, everything works out just fine. So then it says, gee, somewhere in the optimizations, they did an optimization wrong. So one of the first principles of optimization is do it right. And then the second is make it fast. And so sometimes the compiler doesn't that.

It's also the case that sometimes you cannot write code that produces the assembly that you want. And in that case, you can actually write the assembly by hand. Now, it used to be many years ago-- many, many years ago-- that a lot of software was written in assembly. In fact, my first job out of college, I spent about half the time programming in assembly language.

And it's not as bad as you would think. But it certainly is easier to have high-level languages that's for sure. You get lot more done a lot quicker.

And the last reason is reverse engineer. You can figure out what a program does when you only have access to its source, so, for example, the matrix multiplication example that I gave on day 1. You know, we had the overall outer structure, but the inner loop, we could not match the Intel math kernel library code. So what do we do?

We didn't have the source for it. We looked to see what it was doing. We said, oh, is that what they're doing? And then we're able to do it ourselves without having to get the sauce from them. So we reverse engineered what they did? So all those are good reasons.

Now, in this class, we have some expectations. So one thing is, you know, assembly is complicated and you needn't memorize the manual. In fact, the manual has over 1,000 pages. It's like-- but here's what we do expect of you.

You should understand how a compiler implements various C linguistic constructs with x86 instructions. And that's what we'll see in the next lecture. And you should be able to read x86 assembly language with the aid of an architecture manual. And on a quiz, for example, we would give you snippets or explain what the op codes that are being used in case it's not there. But you should have some understanding of that, so you can see what's actually happening.

You should understand the high-level performance implications of common assembly patterns. OK, so what does it mean to do things in a particular way in terms of performance? So some of them are quite obvious. Vector operations tend to be faster than doing the same thing with a bunch of scalar operations.

If you do write an assembly, typically what we use is there are a bunch of compiler intrinsic functions, built-ins, so-called, that allow you to use the assembly language instructions. And you should be after we've done this able to write code from scratch if the situation demands it sometime in the future. We won't do that in this class, but we expect that you will be in a position to do that after-- you should get a mastery to the level where that would not be impossible for you to do. You'd be able to do that with a reasonable amount of effort.

So the rest of the lecture here is I'm going to first start by talking about the instruction set architecture of the x86-64, which is the one that we are using for the cloud machines that we're using. And then I'm going to talk about floating point in vector hardware and then I'm going to do an overview of computer architecture.

Now, all of this I'm doing-- this is software class, right? Software performance engineering we're doing. So the reason we're doing this is so you can write code that better matches the hardware, therefore to better get it.

In order to do that, I could give things at a high-level. My experience is that if you really want to understand something, you want to understand it to level that's necessary and then one level below that. It's not that you'll necessarily use that one level below it, but that gives you insight as to why that layer is what it is and what's really going on. And so that's kind of what we're going to do. We're going to do a dive that takes us one level beyond what you probably will need to know in the class, so that you have a robust foundation for understanding.

Does that makes sense? That's my part of my learning philosophy is you know go one step

beyond. And then you can come back.

The ISA primer, so the ISA talks about the syntax and semantics of assembly. There are four important concepts in the instruction set architecture-- the notion of registers, the notion of instructions, the data types, and the memory addressing modes. And those are sort of indicated. For example, here, we're going to go through those one by one.

So let's start with the registers. So the registers is where the processor stores things. And there are a bunch of x86 registers, so many that you don't need to know most of them. The ones that are important are these.

So first of all, there a general purpose registers. And those typically have width 64. And there are many of those.

There is a so-called flags register, called RFLAGS, which keeps track of things like whether there was an overflow, whether the last arithmetic operation resulted in a zero, whether a kid there was a carryout of a word or what have you.

The next one is the instruction pointer. So the assembly language is organized as a sequence of instructions. And the hardware marches linearly through that sequence, one after the other, unless it encounters a conditional jump or an unconditional jump, in which case it'll branch to whatever the location is. But for the most part, it's just running straight through memory.

Then there are some registers that were added quite late in the game, namely the SSE registers and the AVX registers. And these are vector registers. So the XMM registers were, when they first did vectorization, they used 128 bits. There's also for AVX, there are the YMM registers. And in the most recent processors, which were not using this term, there's another level of AVX that gives you 512-bit registers.

Maybe we'll use that for the final project, because it's just like a little more power for the game playing project. But for most of what you'll be doing, we'll just be keeping to the C4 instances in AWS that you guys have been using.

Now, the x86-64 didn't start out as x86-64. It started out as x86. And it was used for machines, in particular the 80-86, which had a 16-bit word. So really short.

How many things can you index with a 16-bit word? About how many?

AUDIENCE: 65,000.

CHARLES LEISERSON: Yeah, about 65,000. 65,536 words you can address, or bytes. This is byte addressing. So that's 65k bytes that you can address.

How could they possibly use that for machines? Well, the answer is that's how much memory was on the machine. You didn't have gigabytes. So as the machines-- as Moore's law marched along and we got more and more memory, then the words had to become wider to be able to index them. Yeah?

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: Yeah, but here's the thing is if you're building stuff that's too expensive and you can't get memory that's big enough, then if you build a wider word, like if you build a word of 32 bits, then your processor just cost twice as much as the next guy's processor. So instead, what they did is they went along as long as that was the common size, and then had some growth pains and went to 32. And from there, they had some more growth pains and went to 64. OK, those are two separate things.

And, in fact, they did they did some really weird stuff. So what they did in fact is when they made these longer registers, they have registers that are aliased to exactly the same thing for the lower bits. So they can address them either by a byte-- so these registers all have the same-- you can do the lower and upper half of the short word, or you can do the 32-bit word or you can do the 64-bit word.

And that's just like if you're doing this today, you wouldn't do that. You wouldn't have all these registers that alias and such. But that's what they did because this is history, not design. And the reason was because when they're doing that they were not designing for long term.

Now, are we going to go to 128-bit addressing? Probably not. 64 bits address is a spectacular amount of stuff. You know, not quite as many-- 2 to the 64th is what? Is like how many gazillions? It's a lot of gazillions. So, yeah, we're not going to have to go beyond 64 probably.

So here are the general purpose registers. And as I mentioned, they have different names, but they cover the same thing. So if you change eax, for example, that also changes rax. And so you see they originally all had functional purposes. Now, they're all pretty much the same thing, but the names have stuck because of history. Instead of calling them registers 0, register 1, or whatever, they all have these funny names.

Some of them still are used for a particular purpose, like rsp is used as the stack pointer. And rbp is used to point to the base of the frame, for those who remember their 6004 stuff. So anyway, there are a whole bunch of them. And they're different names depending upon which part of the register you're accessing.

Now, the format of an x86-64 instruction code is to have an opcode and then an operand list. And the operand list is typically 0, 1, 2, or rarely 3 operands separated by commas. Typically, all operands are sources and one operand might also be the destination.

So, for example, if you take a look at this add instruction, the operation is an add. And the operand list is these two registers. One is edi and the other is ecx. And the destination is the second one.

When you add-- in this case, what's going on is it's taking the value in ecx, adding the value in edi into it. And the result is in ecx. Yes?

AUDIENCE: Is there a convention for where the destination [INAUDIBLE]

CHARLES LEISERSON: Funny you should ask. Yes. So what does op A, B mean? It turns out naturally that the literature is inconsistent about how it refers to operations. And there's two major ways that are used. One is the AT&T syntax, and the other is the Intel syntax.

So the AT&T syntax, the second operand is the destination. The last operand is the destination. In the Intel syntax, the first operand is the destination. OK, is that confusing?

So almost all the tools that we're going to use are going to use the AT&T syntax. But you will read documentation, which is Intel documentation. It will use the other syntax. Don't get confused. OK? I can't help-- it's like I can't help that this is the way the state of the world is. OK? Yeah?

AUDIENCE: Are there tools that help [INAUDIBLE]

CHARLES LEISERSON: Oh, yeah. In particular, if you could compile it and undo, but I'm sure there's-- I mean, this is not a hard translation thing. I'll bet if you just Google, you can in two minutes, in two seconds, find somebody who will translate from one to the other. This is not a complicated translation process.

Now, here are some very common x86 opcodes. And so let me just mention a few of these,

because these are ones that you'll often see in the code.

So move, what do you think move does?

AUDIENCE: Moves something.

CHARLES LEISERSON: Yeah, it puts something in one register into another register. Of course, when it moves it, this is computer science move, not real move. When I move my belongings in my house to my new house, they're no longer in the old place, right? But in computer science, for some reason, when we move things we leave a copy behind. So they may call it move, but--

AUDIENCE: Why don't they call it copy?

CHARLES LEISERSON: Yeah, why don't they call it copy? You got me. OK, then there's conditional move. So this is move based on a condition-- and we'll see some of the ways that this is-- like move if flag is equal to 0 and so forth, so basically conditional move. It doesn't always do the move.

Then you can extend the sign. So, for example, suppose you're moving from a 32-bit value register into a 64-bit register. Then the question is, what happens to high order bits? So there's two basic mechanisms that can be used. Either it can be filled with zeros, or remember that the first bit, or the leftmost bit as we think of it, is the sign bit from our electron binary. That bit will be extended through the high order part of the word, so that the whole number if it's negative will be negative and if it's positive, it'll be zeros and so forth. Does that makes sense?

Then there are things like push and pop to do stacks. There's a lot of integer arithmetic. There's addition, subtraction, multiplication, division, various shifts, address calculation shifts, rotations, incrementing, decrementing, negating, etc. There's also a lot of binary logic, AND, OR, XOR, NOT. Those are all doing bitwise operations.

And then there is Boolean logic, like testing to see whether some value has a given value or comparing. There's unconditional jump, which is jump. And there's conditional jumps, which is jump with a condition. And then things like subroutines. And there are a bunch more, which the manual will have and which will undoubtedly show up. Like, for example, there's the whole set of vector operations we'll talk about a little bit later.

Now, the opcodes may be augmented with a suffix that describes the data type of the operation or a condition code. OK, so an opcode for data movement, arithmetic, or logic use a single character suffix to indicate the data type. And if the suffix is missing, it can usually be

inferred.

So take a look at this example. So this is a move with a q at the end. What do you think q stands for?

AUDIENCE: Quad words?

CHARLES LEISERSON: Quad word. OK, how many bytes in a quad word?

AUDIENCE: Eight.

CHARLES LEISERSON: Eight. That's because originally it started out with a 16-bit word. So they said a quad word was four of those 16-bit words. So that's 8 bytes. You get the idea, right?

But let me tell you this is all over the x86 instruction set. All these historical things and all these mnemonics that if you don't understand what they really mean, you can get very confused. So in this case, we're moving a 64-bit integer, because a quad word has 8 bytes or 64 bits.

This is one of my-- it's like whenever I prepare this lecture, I just go into spasms of laughter, as I look and I say, oh, my god, they really did that like. For example, on the last page, when I did subtract. So the sub-operator, if it's a two argument operator, it subtracts the-- I think it's the first and the second. But there is no way of subtracting the other way around. It puts the destination in the second one. It basically takes the second one minus the first one and puts that in the second one.

But if you wanted to have it the other way around, to save yourself a cycle-- anyway, it doesn't matter. You can't do it that way. And all this stuff the compiler has to understand.

So here are the x86-64 data types. The way I've done it is to show you the difference between C and x86-64, so for example, here are the declarations in C. So there's a char, a short, int, unsigned int, long, etc. Here's an example of a C constant that does those things. And here's the size in bytes that you get when you declare that.

And then the assembly suffix is one of these things. So in the assembly, it says b or w for a word, an l or d for a double word, a q for a quad word, i.e. 8 bytes, single precision, double precision, extended precision.

So sign extension use two date type suffixes. So here's an example. So the first one says

we're going to move. And now you see I can't read this without my cheat sheet. So what is this saying?

This is saying, we're going to move with a zero-extend. And it's going to be the first operand is a byte, and the second operation is a long. Is that right? If I'm wrong, it's like I got to look at the chart too. And, of course, we don't hold you to that.

But the z there says extends with zeros. And the S says preserve the sign. So that's the things.

Now, that would all be all well and good, except that then what they did is if you do 32-bit operations, where you're moving it to a 64-bit value, it implicitly zero-extends the sign. If you do it for smaller values and you store it in, it simply overwrites the values in those registers. It doesn't touch the high order bits. But when they did the 32 to 64-bit extension of the instruction set, they decided that they wouldn't do what had been done in the past. And they decided that they would zero-extend things, unless there was something explicit to the contrary. You got me, OK.

Yeah, I have a friend who worked at Intel. And he had a joke about the Intel instructions set. You'll discover the Intel instruction set is really complicated. He says, here's the idea of the Intel instruction set. He said, to become an Intel fellow, you need to have an instruction in the Intel instruction set. You have an instruction that you invented and that that's now used in Intel. He says nobody becomes an Intel fellow for removing instructions. So it just sort of grows and grows and grows and gets more and more complicated for each thing.

Now, once again, for extension, you can sign-extend. And here's two examples. In one case, moving an 8-bit integer to a 32-bit integer and zero-extended it versus preserving the sign.

Conditional jumps and conditional moves also use suffixes to indicate the condition code. So here, for example, the ne indicates the jump should only be taken if the argument of the previous comparison are not equal. So ne is not equal. So you do a comparison, and that's going to set a flag in the RFLAGS register. Then the jump will look at that flag and decide whether it's going to jump or not or just continue the sequential execution of the code.

And there are a bunch of things that you can jump on which are status flags. And you can see the names here. There's Carry. There's Parity. Parity is the XOR of all the bits in the word. Adjust, I don't even know what that's for. There's the Zero flag. It tells whether it's a zero. There's a Sign flag, whether it's positive or negative. There's a Trap flag and Interrupt enable

and Direction, Overflow. So anyway, you can see there are a whole bunch of these.

So, for example here, this is going to decrement rbx. And then it sets the Zero flag if the results are equal. And then the jump, the conditional jump, jumps to the label if the ZF flag is not set, in this case. OK, it make sense? After a fashion. Doesn't make rational sense, but it does make sense.

Here are the main ones that you're going to need. The Carry flag is whether you got a carry or a borrow out of the most significant bit. The Zero flag is if the ALU operation was 0, whether the last ALU operation had the sign bit set. And the overflow says it resulted in arithmetic overflow.

The condition codes are-- if you put one of these condition codes on your conditional jump or whatever, this tells you exactly what the flag is that is being set. So, for example, the easy ones are if it's equal. But there are some other ones there.

So, for example, if you say why, for example, do the condition codes e and ne, check the Zero flag? And the answer is typically, rather than having a separate comparison, what they've done is separate the branch from the comparison itself. But it also needn't be a compare instruction. It could be the result of the last arithmetic operation was a zero, and therefore it can branch without having to do a comparison with zero.

So, for example, if you have a loop. where you're decrementing a counter till it gets to 0, that's actually faster by one instruction to compare whether the loop index hits 0 than it is if you have the loop going up to n, and then every time through the loop having to compare with n in order before you can branch.

So these days that optimization doesn't mean anything, because, as we'll talk about in a little bit, these machines are so powerful, that doing an extra integer arithmetic like that probably has no bearing on the overall cost. Yeah?

AUDIENCE: So this instruction doesn't take arguments? It just looks at the flags?

CHARLES LEISERSON: Just looks at the flags, yep. Just looks at the flags. It doesn't take any arguments.

Now, the next aspect of this is you can give registers, but you also can address memory. And there are three direct addressing modes and three indirect addressing modes. At most, one

operand may specify a memory address.

So here are the direct addressing modes. So for immediate what you do is you give it a constant, like 172, random constant, to store into the register, in this case. That's called an immediate. What happens if you look at the instruction, if you look at the machine language, 172 is right in the instruction. It's right in the instruction, that number 172.

Register says we'll move the value from the register, in this case, %cx. And then the index of the register is put in that part.

And direct memory says use a particular memory location. And you can give a hex value. When you do direct memory, it's going to use the value at that place in memory. And to indicate that memory is going to take you, on a 64-bit machine, 64 8-bytes to specify that memory. Whereas, for example, the move q, 172 will fit in 1 byte. And so I'll have spent a lot less storage in order to do it. Plus, I can do it directly from the instruction stream. And I avoid having an access to memory, which is very expensive.

So how many cycles does it take if the value that you're fetching from memory is not in cache or whatever or a register? If I'm fetching something from memory, how many cycles of the machine does it typically take these days. Yeah.

AUDIENCE: A few hundred?

CHARLES LEISERSON: Yeah, a couple of hundred or more, yeah, a couple hundred cycles. To fetch something from memory. It's so slow. No, it's the processors are so fast.

And so clearly, if you can get things into registers, most registers you can access in a single cycle. So we want to move things close to the processor, operate on them, shove them back. And while we pull things from memory, we want other things to be to be working on. And so the hardware is all organized to do that.

Now, of course, we spend a lot of time fetching stuff from memory. And that's one reason we use caching. And we'll have a big thing-- caching is really important. We're going spend a bunch of time on how to get the best out of your cache.

There's also indirect addressing. So instead of just giving a location, you say, oh, let's go to some other place, for example, a register, and get the value and the address is going to be stored in that location. So, for example here, register indirect says, in this case, move the

contents of rax into-- sorry, the contents is the address of the thing that you're going to move into rdi. So if rax was location 172, then it would take whatever is in location 172 and put it in rdi.

Registered index says, well, do the same thing, but while you're at it, add an offset. So once again, if rax had 172, in this case it would go to 344 to fetch the value out of that location 344 for this particular instruction.

And then instruction-pointer relative, instead of indexing off of a general purpose register, you index off the instruction pointer. That usually happens in the code where the code is-- for example, you can jump to where you are in the code plus four instructions. So you can jump down some number of instructions in the code.

Usually, you'll see that only with use with control, because you're talking about things. But sometimes they'll put some data in the instruction stream. And then it can index off the instruction pointer to get those values without having to soil another register.

Now, the most general form is base indexed scale displacement addressing. Wow. This is a move that has a constant plus three terms. And this is the most complicated instruction that is supported.

The mode refers to the address whatever the base is. So the base is a general purpose register, in this case, rdi. And then it adds the index times the scale. So the scale is 1, 2, 4, or 8. And then a displacement, which is that number on the front. And this gives you very general indexing of things off of a base point.

You'll often see this kind of accessing when you're accessing stack memory, because everything you can say, here is the base of my frame on the stack, and now for anything that I want to add, I'm going to be going up a certain amount. I may scaling by a certain amount to get the value that I want. So once again, you will become familiar with a manual. You don't have to memorize all these, but you do have to understand that there are a lot of these complex addressing modes.

The jump instruction take a label as their operand, which identifies a location in the code. For this, the labels can be symbols. In other words, you can say here's a symbol that I want to jump to. It might be the beginning of a function, or it might be a label that's generated to be at the beginning of a loop or whatever. They can be exact addresses-- go to this place in the

code. Or they can be relative address-- jump to some place as I mentioned that's indexed off the instruction pointer.

And then an indirect jump takes as its operand an indirect address-- oop, I got-- as its operand as its operand. OK, so that's a typo. It just takes an operand as an indirect address. So basically, you can say, jump to whatever is pointed to by that register using whatever indexing method that you want.

So that's kind of the overview of the assembly language. Now, let's take a look at some idioms. So the XOR opcode computes the bitwise XOR of A and B. We saw XOR was a great trick for swapping numbers, for example, the other day.

So often in the code, you will see something like this, xor rax rax. What does that do? Yeah.

AUDIENCE: Zeros the register.

CHARLES LEISERSON: It zeros the register. Why does that zero the register?

AUDIENCE: Is the XOR just the same?

CHARLES LEISERSON: Yeah, it's basically taking the results of rax, the results rax, xor-ing them. And when you XOR something with itself, you get zero, storing that back into it. So that's actually how you zero things. So you'll see that.

Whenever you see that, hey, what are they doing? They're zeroing the register. And that's actually quicker and easier than having a zero constant that they put into the instruction. It saves a byte, because this ends up being a very short instruction. I don't remember how many bytes that instruction is.

Here's another one, the test opcode, test A, B, computes the bitwise AND of A and B and discards the result, preserving the RFLAGS register. So basically, it says, what does the test instruction for these things do? So what is the first one doing? So it takes rcx-- yeah.

AUDIENCE: Does it jump? It jumps to [INAUDIBLE] rcx [INAUDIBLE]

So it takes the bitwise AND of A and B. And so then it's saying jump if equal. So--

AUDIENCE: An AND would be non-zero in any of the bits set.

CHARLES LEISERSON: Right. AND is non-zero if any of the bits are set.

AUDIENCE: Right. So if the zero flag were set, that means that rcx was zero.

CHARLES LEISERSON: That's right. So if the Zero flag is set, then rcx is set. So this is going to jump to that location if rcx holds the value 0. In all the other cases, it won't set the Zero flag because the result of the AND will be 0. So once again, that's kind of an idiom that they use.

What about the second one? So this is a conditional move. So both of them are basically checking to see if the register is 0. And then doing something if it is or isn't. But those are just idioms that you sort of have to look at to see how it is that they accomplish their particular thing.

Here's another one. So the ISA can include several no-op, no operation instructions, including nop, nop A-- that's an operation with an argument-- and data16, which sets aside 2 bytes of a nop. So here's a line of assembly that we found in some of our code-- data16 days16 data16 nopw and then %csx. So nopw is going to take this argument, which has got all this address calculation in it.

So what do you think this is doing? What's the effect of this, by the way? They're all no-ops. So the effect is? Nothing. The effect is nothing. OK, now it does set the RFLAGS. But basically, mostly, it does nothing.

Why would a compiler generate assembly with these idioms? Why would you get that kind of-- that's crazy, right? Yeah.

AUDIENCE: Could it be doing some cache optimization?

CHARLES LEISERSON: Yeah, it's actually doing alignment optimization typically or code size. So it may want to start the next instruction on the beginning of a cache line. And, in fact, there's a directive to do that. If you want all your functions to start at the beginning of cache line, then it wants to make sure that if code gets to that point, you'll just proceed to jump through memory, continue through memory. So mainly is to optimize memory. So you'll see those things. I mean, you just have to realize, oh, that's the compiler generating some sum no-ops.

So that's sort of our brief excursion over assembly language, x86 assembly language. Now, I want to dive into floating-point and vector hardware, which is going to be the main part. And

then if there's any time at the end, I'll show the slides-- I have a bunch of other slides on how branch prediction works and a variety of other machines sorts of things, that if we don't get to, it's no problem. You can take a look at the slides, and there's also the architecture manual.

So floating-point instruction sets, so mostly the scalar floating-point operations are access via couple of different instruction sets. So the history of floating point is interesting, because originally the 80-86 did not have a floating-point unit. Floating-point was done in software. And then they made a companion chip that would do floating-point. And then they started integrating and so forth as miniaturization took hold.

So the SSE and AVX instructions do both single and double precision scalar floating-point, i.e. floats or doubles. And then the x86 instructions, the x87 instructions-- that's the 80-87 that was attached to the 80-86 and that's where they get them-- support single, double, and extended precision scalar floating-point arithmetic, including float double and long double. So you can actually get a great big result of a multiply if you use the x87 instruction sets. And they also include vector instructions, so you can multiply or add there as well-- so all these places on the chip where you can decide to do one thing or another.

Compilers generally like the SSE instructions over the x87 instructions because they're simpler to compile for and to optimize. And the SSE opcodes are similar to the normal x86 opcodes. And they use the XMM registers and floating-point types. And so you'll see stuff like this, where you've got a movesd and so forth. The suffix there is saying what the data type. In this case, it's saying it's a double precision floating-point value, i.e. a double.

Once again, they're using suffix. The sd in this case is a double precision floating-point. The other option is the first letter says whether it's single, i.e. a scalar operation, or packed, i.e. a vector operation. And the second letter says whether it's single or double precision. And so when you see one of these operations, you can decode, oh, this is operating on a 64-bit value or a 32-bit value, floating-point value, or on a vector of those values.

Now, what about these vectors? So when you start using the packed representation and you start using vectors, you have to understand a little bit about the vector units that are on these machines. So the way a vector unit works is that there is the processor issuing instructions. And it issues the instructions to all of the vector units.

So for example, if you take a look at a typical thing, you may have a vector width of four vector units. Each of them is often called a lane-- l-a-n-e. And the x is the vector width. And so when

the instruction is given, it's given to all of the vector units. And they all do it on their own local copy of the register.

So the register you can think of as a very wide thing broken into several words. And when I say add two vectors together, it'll add four words together and store it back into another vector register. And so whatever k is-- in the example I just said, k was 4. And the lanes are the thing that each of which contains the integer floating-point arithmetic.

But the important thing is that they all operate in lock step. It's not like one is going to do one thing and another is going to do another thing. They all have to do exactly the same thing. And the basic idea here is for the price of one instruction, I can command a bunch of operations to be done.

Now, generally, vector instructions operate in an element-wise fashion, where you take the i-th element of one vector and operate on it with the i-th element of another vector. And all the lanes perform exactly the same operation. Depending upon the architecture, some architectures, the operands need to be aligned. That is you've got to have the beginnings at the exactly same place in memory, a multiple of the vector length. There are others where the vectors can be shifted in memory.

Usually, there's a performance difference between the two. If it does support-- some of them will not support unaligned vector operations. So if it can't figure out that they're aligned, I'm sorry, your code will end up being executed scalar, in a scalar fashion. If they are aligned, it's got to be able to figure that out. And in that case-- sorry, if it's not aligned, but you do support vector operizations unaligned, it's usually slower than if they are aligned. And for some machines now, they actually have good performance on both. So it really depends upon the machine.

And then also there are some architectures will support cross-lane operation, such as inserting or extracting subsets of vector elements, permuting, shuffling, scatter, gather types of operations.

So x86 supports several instruction sets, as I mentioned. There's SSE. There's AVX. There's AVX2. And then there's now the AVX-512, or sometimes called AVX3, which is not available on the machines that we'll be using, the Haswell machines that we'll be doing.

Generally, the AVX and AVX2 extend the SSE instruction set by using the wider registers and

operate on a 2. The SSE use wider registers and operate on at most two operands. The AVX ones can use the 256 and also have three operands, not just two operations. So say you can say add A to B and store it in C, as opposed to saying add A to B and store it in B. So it can also support three.

Yeah, most of them are similar to traditional opcodes with minor differences. So if you look at them, if you have an SSE, it basically looks just like the traditional name, like add in this case, but you can then say, do a packed add or a vector with packed data. So the v prefix it's AVX. So if you see it's v, you go to the part in the manual that says AVX. If you see the p's, that say it's packed data. Then you go to SSE if it doesn't have the v.

And the p prefix distinguishing integer vector instruction, you got me. I tried to think why is p distinguishing an integer? It's like p, good mnemonic for integer, right?

Then in addition, they do this aliasing trick again, where the YMM registers actually alias the XMM registers. So you can use both operations, but you've got to be careful what's going on, because they just extended them. And now, of course, with AVX-512, they did another extension to 512 bits.

That's vectors stuff. So you can use those explicitly. The compiler will vectorize for you. And the homework this week takes you through some vectorization exercises. It's actually a lot of fun. We were just going over it in a staff meeting. And it's really fun. I think it's a really fun exercise. We introduced that last year, by the way, or maybe two years ago. But, in any case, it's a fun one-- for my definition of fun, which I hope is your definition of fun.

Now, I want to talk generally about computer architecture. And I'm not going to get through all of these slides, as I say. But I want to get started on the and give you a sense of other things going on in the processor that you should be aware of.

So in 6.004, you probably talked about a 5-stage processor. Anybody remember that? OK, 5-stage processor. There's an Instruction Fetch. There's an Instruction Decode. There's an Execute. Then there's a Memory Addressing. And then you Write back the values.

And this is done as a pipeline, so as to make-- you could do all of this in one thing, but then you have a long clock cycle. And you'll only be able to do one thing at a time. Instead, they stack them together.

So here's a block diagram of the 5-stage processor. We read the instruction from memory in the instruction fetch cycle. Then we decode it. Basically, it takes a look at, what is the opcode, what are the addressing modes, et cetera, and figures out what it actually has to do and actually performs the ALU operations. And then it reads and writes the data memory. And then it writes back the results into registers. That's typically a common way that these things go for a 5-stage processor.

By the way, this is vastly oversimplified. You can take 6823 if you want to learn truth. I'm going to tell you nothing but white lies for this lecture.

Now, if you look at the Intel Haswell, the machine that we're using, it actually has between 14 and 19 pipeline stages. The 14 to 19 reflects the fact that there are different paths through it that take different amounts of time. It also I think reflects a little bit that nobody has published the Intel internal stuff. So maybe we're not sure if it's 14 to 19, but somewhere in that range. But I think it's actually because the different lengths of time as I was explaining.

So what I want to do is-- you've seen the 5-stage price line. I want to talk about the difference between that and a modern processor by looking at several design features. We already talked about vector hardware. I then want to talk about super scalar processing, out of order execution, and branch prediction a little bit. And the out of order, I'm going to skip a bunch of that because it has to do with score boarding, which is really interesting and fun, but it's also time consuming. But it's really interesting and fun. That's what you learn in 6823.

So historically, there's two ways that people make processors go faster-- by exploiting parallelism and by exploiting locality. And parallelism, there's instruction-- well, we already did word-level parallelism in the bit tricks thing. But there's also instruction-level parallelism, so-called ILB, vectorization and multicore. And for locality, the main thing that's used there is caching.

I would say also the fact that you have a design with registers that also reflects locality, because the way that the processor wants to do things is fetch stuff from memory. It doesn't want to operate on it in memory. That's very expensive. It wants to fetch things into memory, get enough of them there that you can do some calculations, do a whole bunch of calculations, and then put them back out there.

So this lecture we're talking about ILP and vectorization. So let me talk about instruction-level parallelism.

So when you have, let's say, a 5-stage pipeline, you're interested in finding opportunities to execute multiple instruction simultaneously. So in instruction 1, it's going to do an instruction fetch. Then it does its decode. And so it takes five cycles for this instruction to complete.

So ideally what you'd like is that you can start instruction 2 on cycle 2, instruction 3 on cycle 3, and so forth, and have 5 instructions-- once you get into the steady state, have 5 instructions executing all the time. That would be ideal, where each one takes just one thing. So that's really pretty good.

And that would improve the throughput. Even though it might take a long time to get one instruction done, I can have many instructions in the pipeline at some time. So each pipeline is executing a different instruction.

However, in practice this isn't what happens. In practice, you discover that there are what's called pipeline stalls. When it comes time to execute an instruction, for some correctness reason, it cannot execute the instruction. It has to wait. And that's a pipeline stall. That's what you want to try to avoid and the compiler tries to Bruce code that will avoid stalls.

So why do stalls happen? They happen because of what are called hazards. There's actually two notions of hazard. And this is one of them. The other is a race condition hazard. This is dependency hazard. But people call them both hazards, just like they call the second stage of compilation compiling. It's like they make up these words.

So here's three types of hazards that can prevent an instruction from executing. First of all, there's what's called a structural hazard. Two instructions attempt to use the same functional unit, the same time. If there's, for example, only one floating-point multiplier and two of them try to use it at the same time, one has to wait. In modern processors, there's a bunch of each of those. But if you have k functional units and k plus 1 instructions want to access it, you're out of luck. One of them is going to have to wait.

The second is a data hazard. This is when an instruction depends on the result of a prior instruction in the pipeline. So one instruction is computing a value that is going to stick in rcx, say. So they stick it into rcx. The other one has to read the value from rcx and it comes later. That other instruction has to wait until that value is written there before it can read it. That's a data hazard.

And a control hazard is where you decide that you need to make a jump and you can't execute the next instruction, because you don't know which way the jump is going to go. So if you have a conditional jump, it's like, well, what's the next instruction after that jump? I don't know. So I have to wait to execute that. I can't go ahead and do the jump and then do the next instruction after it, because I don't know what happened to the previous one.

Now of these, we're going to mostly talk about data hazards. So an instruction can create a data hazard-- I can create a data hazard due to a dependence between i and j. So the first type is called a true dependence, or I read after write dependence.

And this is where, as in this example, I'm adding something and storing into rax and the next instruction wants to read from rax. So the second instruction can't get going until the previous one or it may stall until the result of the previous one is known.

There's another one called an anti-dependence. This is where I want to write into a location, but I have to wait until the previous instruction has read the value, because otherwise I'm going to clobber that instruction and clobber the value before it gets read. so that's an anti-dependence.

And then the final one is an output dependence, where they're both trying to move something to are rax. So why would two things want to move things to the same location? After all, one of them is going to be lost and just not do that instruction. Why wouldn't--

AUDIENCE: Set some flags.

CHARLES LEISERSON: Yeah, maybe because it wants to set some flags. So that's one reason that it might do this, because you know the first instruction set some flags in addition to moving the output to that location. And there's one other reason. What's the other reason? I'm blanking. There's two reasons. And I didn't put them in my notes. I don't remember. OK, but anyway, that's a good question for quiz then. OK, give me two reasons-- yeah.

AUDIENCE: Can there be intermediate instructions like between those [INAUDIBLE]

CHARLES LEISERSON: There could, but of course then if it's going to use that register, then-- oh, I know the other reason. So this is still good for a quiz.

The other reason is there may be aliasing going on. Maybe an intervening instruction uses one of the values in its aliasist. So uses part of the result or whatever, there still could be a

dependency.

Anyway, some arithmetic operations are complex to implement in hardware and have long latencies. So here's some sample opcodes and how many latency they take. They take a different number. So, for example, integer division actually is variable, but a multiply takes about three times what most of the integer operations are. And floating-point multiply is like 5. And then fma, what's fma? Fused multiply add. This is where you're doing both a multiply and an add. And why do we care about fuse multiply adds?

AUDIENCE: For memory accessing and [INAUDIBLE]

CHARLES LEISERSON: Not for memory accessing. This is actually floating-point multiply and add. It's called linear algebra. So when you do major multiplication, you're doing dot product. You're doing multiplies and adds. So that kind of thing, that's where you do a lot of those.

So how does the hardware accommodate these complex operations? So the strategy that much hardware tends to use is to have separate functional units for complex operations, such as floating-point arithmetic. So there may be in fact separate registers, for example, the XMM registers, that only work with the floating point.

So you have your basic 5-stage pipeline. You have another pipeline that's off on the side. And it's going to take multiple cycles sometimes for things and maybe pipeline to a different depth. And so you basically separate these operations. The functional units may be pipelined, fully, partially, or not at all. And so I now have a whole bunch of different functional units, and there's different paths that I'm going to be able to take through the data path of the processor.

So in Haswell, they have integer vector floating-point distributed among eight different ports, which is sort from the entry. So given that, things get really complicated. If we go back to our simple diagram, suppose we have all these additional functional units, how can I now exploit more instruction-level parallelism? So right now, we can start up one operation at a time. What might I do to get more parallelism out of the hardware that I've got? What do you think computer architects did? OK.

AUDIENCE: It's a guess but, you could glue together [INAUDIBLE]

CHARLES LEISERSON: Yeah, so even simpler than that, but which is implied in what you're saying, is you can just fetch and issue multiple instructions per cycle. So rather than just doing one per cycle as we showed with a typical pipeline processor, let me fetch several that use different parts of the

processor pipeline, because they're not going to interfere, to keep everything busy. And so that's basically what's called a super scalar processor, where it's not executing one thing at a time. It's executing multiple things at a time.

So Haswell, in fact, breaks up the instructions into simpler operations, called micro-ops. And they can emit for micro-ops per cycle to the rest of the pipeline. And the fetch and decode stages implement optimizations on micro-op processing, including special cases for common patents.

For example, if it sees the XOR of rax and rax, it knows that rax is being set to 0. It doesn't even use a functional unit for that. It just does it and it's done. It has just a special logic that observes that because it's such a common thing to set things out. And so that means that now your processor can execute a lot of things at one time. And that's the machines that you're doing.

That's why when I said if you save one add instruction, it probably doesn't make any difference in today's processor, because there's probably an idle adder lying around. There's probably a-- did I read caught how many-- where do we go here? Yeah, so if you look here, you can even discover that there are actually a bunch of ALUs that are capable of doing an add. So they're all over the map in Haswell.

Now, still, we are insisting that the processors execute in things in order. And that's kind of the next stage is, how do you end up making things run-- that is, how do you make it so that you can free yourself from the tyranny of one instruction after the other? And so the first thing is there's a strategy called bypassing.

So suppose that you have instructions running into rax. And then you're going to use that to read. Well, why bother waiting for it to be stored into the register file and then pulled back out for the second instruction? Instead, let's have a bypass, a special circuit that identifies that kind of situation and feeds it directly to the next instruction without requiring that it go into the register file and back out. So that's called bypassing. There are lots of places where things are bypassed. And we'll talk about it more.

So normally, you would stall waiting for it to be written back. And now, when you eliminate it, now I can move it way forward, because I just use the bypass path to execute. And it allows the second instruction to get going earlier.

What else can we do? Well, let's take a large code example. Given the amount of time, what I'm going to do is basically say, you can go through and figure out what are the read after write dependencies and the write after read dependencies. They're all over the place. And what you can do is if you look at what the dependencies are that I just flashed through, you can discover, oh, there's all these things. Each one right now has to wait for the previous one before it can get started.

But there are some-- for example, the first one is just issue order. You can't start the second-- if it's in order, you can't start the second till you've started the first, that it's finished the first stage. But the other thing here is there's a data dependence between the second and third instructions. So if you look at the second and third instructions, they're both using XMM2. And so we're prevented.

So one of the questions there is, well, why not do a little bit better by taking a look at this as a graph and figuring out what's the best way through the graph? And there are a bunch of tricks you can do there, which I'll run through here very quickly. And you can take a look at these.

You can discover that some of these dependencies are not real dependence. And as long as you're willing to execute things out of order and keep track of that, it's perfectly fine. If you're not actually dependent on it, then just go ahead and execute it. And then you can advance things.

And then the other trick you can use is what's called register renaming. If you have a destination that's going to be read from-- sorry, if I want to write to something, but I have to wait for something else to read from it, the write after read dependence, then what I can do is just rename the register, so that I have something to write to that is the same thing. And there's a very complex mechanism called score boarding that does that.

So anyway, you can take a look at all of these tricks. And then the last thing that I want to-- so this is this part I was going to skip over. And indeed, I don't have time to do it.

I just want to mention the last thing, which is worthwhile. So this-- you don't have to know any of the details of that part. But it's in there if you're interested. So it does renaming and reordering.

And then the last thing I do want to mention is branch prediction. So when you come to branch prediction, the outcome, you can have a hazard because the outcome is known too late. And

so in that case, what they do is what's called speculative execution, which you've probably heard of.

So basically that says I'm going to guess the outcome of the branch and execute. If it's encountered, you assume it's taken and you execute normally. And if you're right, everything is hunky dory. If you're wrong, it cost you something like a-- you have to undo that speculative computation and the effect is sort of like stalling. So you don't want that to happen.

And so a mispredicted branch on Haswell costs about 15 to 20 cycles. Most of the machines use a branch predictor to tell whether or not it's going to do. There's a little bit of stuff here about how you tell about whether a branch is going to be predicted or not. And you can take a look at that on your own.

So sorry to rush a little bit the end, but I knew I wasn't going to get through all of this. But it's in the notes, in the slides when we put it up. And this is really kind of interesting stuff.

Once again, remember that I'm dealing with this at one level below what you really need to do. But it is really helpful to understand that layer so you have a deep understanding of why certain software optimizations work and don't work. Sound good? OK, good luck on finishing your project 1's.