**JULIAN SHUN:** Good afternoon, everyone. So today, we have TB Schardl here. He's going to give us the lecture on C to assembly. So TB's a research scientist here at MIT working with Charles Leiserson. He also taught this class with me last year, and he got one of the best ratings ever for this class. So I'm really looking forward to his lecture.

**TAO SCHARDL:** All right, great. So thank you for the introduction, Julian. So I hear you just submitted the beta for project 1. Hopefully, that went pretty well. How many of you slept in the last 24 hours? OK, good. All right, so it went pretty well. That sounds great.

Yeah, so today, we're going to be talking about C to assembly. And this is really a continuation from the topic of last lecture, where you saw computer architecture, if I understand correctly. Is that right? You looked at computer architecture, x86-64 assembly, that sort of thing.

So how many of you walked away from that lecture thinking, oh yeah, x86-64 assembly, this is easy? This is totally intuitive. Everything makes perfect sense. There's no weirdness going on here whatsoever.

How many of you walked away not thinking that? Thinking that perhaps this is a little bit strange, this whole assembly language. Yeah, I'm really in the later cab. x86 is kind of a strange beast. There are things in there that make no sense. Quad word has 8 bytes. P stands for integer, that sort of thing.

So when we move on to the topic of seeing how C code gets translated into assembly, we're translating into something that's already pretty complicated. And the translation itself isn't going to be that straightforward. So we're going to have to find a way to work through that. And I'll outline the strategy that we'll be using in the start of this presentation.

But first, let's quickly review. Why do we care about looking at assembly? You should have seen this slide from the last lecture. But essentially, assembly is a more precise representation of the program than the C code itself. And if you look at the assembly, that can reveal details about the program that are not obvious when you just look at the C code directly.

There are implicit things going on in the C code, such as type cast or the usage of registers versus memory on the machine. And those can have performance implications. So it's valuable to take a look at the assembly code directly.

It can also reveal what the compiler did or did not do when it tried to optimize the program. For example, you may have written a division operation or a multiply operation. But somehow, the compiler figured out that it didn't really need to do a divide or multiply to implement that operation. It could implement it more quickly using simpler, faster operations, like addition and subtraction or shift. And you would be able to see that from looking at the assembly.

Bugs can also arise only at a low level. For example, there may be a bug in the program that only creates unexpected behavior when you optimize the code at 03. So that means, when you're debugging and with that OG or -01, you wouldn't see any unusual behaviors. But when you crank up the optimization level, suddenly, things start to fall apart. Because the C code itself didn't change, it can be hard to spot those bugs. Looking at the assembly can help out in that regard.

And when worse comes to worse, if you really want to make your code fast, it is possible to modify the assembly code by hand. One of my favorite uses of looking at the assembly, though, is actually reverse engineering. If you can read the assembly for some code, you can actually decipher what that program does, even when you only have access to the binary of that program, which is kind of a cool thing.

It takes some practice to read assembly at that level. One trick that some of us in Professor Leiserson's research group have used in the past to say figure out what Intel's Math Kernel Library is doing to multiply matrices.

Now, as I mentioned before, at the end of last lecture, you saw some computer architecture. And you saw the basics of x86-64 assembly, including all the stuff, like the instructions, the registers, the various data types, memory addressing modes, the RFLAGS registered with those condition codes, and that sort of thing. And today, we want to talk about how C code gets implemented in that assembly language.

OK, well, if we consider how C code becomes assembly and what that process actually looks like, we know that there is a compiler involved. And the compiler is a pretty sophisticated piece of software. And, frankly, the compiler has a lot of work to do in order to translate a C program into assembly.

For example, it has to choose what assembly instructions are going to be used to implement those C operations. It has to implement C conditionals and loops-- those if, then, elses and those for and why loops-- into jumps and branches. It has to choose registers and memory locations to store all of the data in the program.

It may have to move data among the registers and the memory locations in order to satisfy various data dependencies. It has to coordinate all the function calls that happen when subroutine A calls B and calls C, and then returns, and so on and so forth. And on top of that, these days, we expect our compiler to try really hard to make that code fast.

So that's a lot of work that the compiler has to do. And as a result, if we take a look at the assembly for any arbitrary piece of C code, the mapping from that C code to the assembly is not exactly obvious, which makes it hard to execute this particular lecture and hard to, in general, read the binary or the assembly for some program and figure out what's really going on.

So what we're going to do today to understand this translation process is we're going to take a look at how that compiler actually reasons about translating C code into assembly. Now this is not a compiler class. 6172 is not a class you take if you want to learn how to build a compiler. And you're not going to need to know everything about a compiler to follow today's lecture. But what we will see is just a little bit about how the compiler understands a program and, later on, how the compiler can translate that program into assembly code.

Now when a compiler compiles a program, it does so through a sequence of stages, which are illustrated on this slide. Starting from the C code, it first pre-processes that code, dealing with all the macros. And that produces a pre-process source. Then the compiler will translate that source code into an intermediate representation. For the client compiler that you're using, that intermediate representation is called LLVM IR. LLVM being the name of the underlying compiler, and IR being the creative name for the intermediate representation.

That LLVM IR is really a sort of pseudo-assembly. It's kind of like assembly, but as we'll see, it's actually a lot simpler than x86-64 assembly. And that's why we'll use it to understand this translation process.

Now it turns out that the compiler does a whole lot of work on that intermediate representation. We're not going to worry about that today. We'll just skip to the end of this pipeline when the compiler translates LLVM IR into assembly code.

Now the nice thing about taking a look at the LLVM IR is that If you're curious, you can actually follow along with the compiler. It is possible to ask clang to compile your code and give you the LLVM IR rather than the assembly. And the flags to do that are somewhat familiar.

Rather than passing the dash s flag, which, hopefully, you've already seen, that will translate C code directly into assembly. If you pass dash s dash omit LLVM, that will produce the LLVM IR. You can also ask clang to translate LLVM IR itself directly into assembly code, and that process is pretty straightforward. You just use the dash S flag once again.

So this is the outline of today's lecture. First, we're going to start with a simple primer on LLVM IR. I know that LLVM IR sounds like another language. Oh, gosh, we have to learn another language. But don't worry. This primer, I would say, is simpler than the x86-64 primer. Based on the slides, for x86-64, that primer was 20-some slides long. This primer is six slides, so maybe a little over a quarter.

Then we'll take a look at how the various constructs in the C programming language get translated into LLVM IR, including straight line code, C functions, conditionals-- in other words, if, then, else-- loops. And we'll conclude that section with just a brief mention of LLVM IR attributes.

And finally, we'll take a look at how LLVM IR gets translated into assembly. And for that, we'll have to focus on what's called the Linux x86-64 calling convention. And we'll conclude with a case study, where we see how this whole process works on a very simple code to compute Fibonacci numbers. Any questions so far?

All right, let's get started. Brief primer on LLVM IR-- so I've shown this in smaller font on some previous slides, but here is a snippet of LLVM IR code. In particular, this is one function within an LLVM IR file. And just from looking at this code, we can see a couple of the basic components of LLVM IR.

In LLVM IR, we have functions. That's how code is organized into these chunks-- chunks called functions. And within each function, the operations of the function are encoded within instructions. And each instruction shows up, at least on this slide, on a separate line.

Those functions operate on what are called LLVM IR registers. These are kind of like the variables. And each of those variables has some associated type. So the types are actually explicit within the IR. And we'll take a look at the types in more detail in a couple of slides.

So based on that high-level overview, we can do a little bit of a comparison between LLVM IR and assembly language. The first thing that we see is that it looks kind of similar to assembly, right? It still has a simple instruction format. There is some destination operand, which we are calling a register. And then there is an equal sign and then an op code, be it add, or call, or what have you, and then some list of source operations. That's roughly what each instruction looks like.

We can also see that the LLVM IR code, it'll turn out. The LLVM IR code adopts a similar structure to the assembly code itself. And control flow, once again, is implemented using conditional branches, as well as unconditional branches.

But one thing that we'll notice is that LLVM IR is simpler than assembly. It has a much smaller instruction set. And unlike assembly language, LLVM IR supports an infinite number of registers. If you can name it, it's a register.

So in that sense, LLVM's notion of registers is a lot closer to C's notion of variables. And when you read LLVM IR, and you see those registers, you should just think about C variables. There's no implicit RFLAGS register, and there no implicit condition codes going on. Everything is pretty explicit in terms of the LLVM.

There's no explicit stack pointer or frame pointer. There's a type system that's explicit in the IR itself. And it's C like in nature, and there are C-like functions for organizing the code overall.

So let's take a look at each of these components, starting with LLVM IR registers. This is basically LLVM's name for a variable. All of the data in LLVM IR is stored in these variables, which are called registers. And the syntax is a percent symbol followed by a name. So %0, %1, %2, that sort of thing.

And as I mentioned before, LLVM IR registers are a lot like c variables. LLVM supports an infinite number of these things, and each distinct register is just distinguished by its name. So %0 is different from %1, because they have different names.

Register names are also local to each LLVM IR function. And in this regard, they're also similar to C variables. If you wrote a C program with two functions, A and B, and each function had a local variable apple, those are two different apples. The apple in A is not the same thing as the apple in B. Similarly, if you had two different LLVM IR functions, and they both described some register five, those are two different variables. They're not automatically aliased.

So here's an example of an LLVM IR snippet. And what we've done here is just highlighted all of the registers. Some of them are being assigned, because they're on the left-hand side of an equal symbol. And some of them are being used as arguments when they show up on the right-hand side.

There is one catch, which we'll see later on, namely that the syntax for LLVM registers ends up being hijacked when LLVM needs to refer to different basic blocks. We haven't defined basic blocks yet. We'll see what that's all about in just a couple of slides. Everyone good so far?

So LLVM IR code is organized into instructions, and the syntax for these instructions is pretty straightforward. We have a register name on the left-hand side, then an equal symbol, and then and op code, followed by an operand list. For example, the top highlight instruction has register six equal to add of sum arguments. And we'll see a little bit more about those arguments later.

That's the syntax for when an instruction actually returns some value. So addition returns the sum of the two operands. Other instructions don't return a value, per se, not a value that you'd store in a local register. And so the syntax for those instructions is just an op code followed by a list of operands. Ironically, the return instruction that you'd find at the end of a function doesn't assign a particular register value. And of course, the operands can be either registers, or constants, or, as we'll see later on, they can identify basic blocks within the function.

The LLVM IR instruction set is smaller than that of x86. x86 contains hundreds of instructions when you start counting up all the vector instructions. And LLVM IR is far more modest in that regard. There's some instructions for data movements, including stack allocation, reading memory, writing memory, converting between types. Yeah, that's pretty much it.

There are some instructions for doing arithmetic or logic, including integer arithmetic, floating-point arithmetic, Boolean logic, binary logic, or address calculations. And then there are a couple of instructions to do control flow. There are unconditional branches or jumps, conditional branches or jumps, subroutines-- that's call or return-- and then there's this magical phi function, which we'll see more of later on in these slides.

Finally, as I mentioned before, everything in LLVM IR is explicitly typed. It's a strongly-typed language in that sense. And the type system looks something like this. For integers, whenever there's a variable of an integer type, you'll see an i followed by some number. And that number defines the number of bits in that integer.

So if you see a variable of type i64, that means it's a 64-bit integer. If you see a variable of type i1, that would be a 1-bit integer or, in other words, a Boolean value. There are also floating-point types, such as double and float.

There are pointer types, when you follow an integer or floating-point type with a star, much like in C, you can have a raise. And that uses a square bracket notation, where, within the square brackets, you'll have some number and then times and then some other type. Maybe it's a primitive type, like an integer or a floating-point. Maybe it's something more complicated.

You can have structs with an LLVM IR. And that uses squiggly brackets with types enumerated on the inside. You can have vector types, which uses angle brackets and otherwise adopts a similar syntax to the array type. Finally, you can occasionally see a variable, which looks like an ordinary register, except that its type is label. And that actually refers to a basic block.

Those are the basic components of LLVM IR. Any questions so far? Everything clear? Everything unclear?

**STUDENT:** What's the basic [INAUDIBLE]?

**TAO SCHARDL:** That should be unclear, and we'll talk about it. Yeah?

**STUDENT:** Is the vector notation there for the vectorization that's done, like the special register is used?

**TAO SCHARDL:** Is the vector notation used for the vector registers? In a sense, yes. The vector operations with an LLVM don't look like SEC or AVX, per se. They look more like ordinary operations, except those ordinary operations work on a vector type. So that's how the vector operations show up in LLVM IR. That make some sense? Cool. Anything else?

OK, that's the whole primer. That's pretty much all of the language that you're going to need to know, at least for this slide deck. We'll cover some of the details as we go along. Let's start translating C code into LLVM IR. Is that good?

All right, let's start with pretty much the simplest thing we can-- straight line C code. What do I mean by straight line C code? I mean that this is a blob of C code that contains no conditionals or loops. So it's just a whole sequence of operations. And that sequence of operations in C code turns into a sequence of operations in LLVM IR.

So in this example here, we have foo of n minus 1 plus bar of n minus 2. That is a sequence of operations. And it turns into the LLVM IR on the right. We can see how that happens.

There are a couple rules of thumb when reading straight line C code and interpreting it in the IR. Arguments to any operation are evaluated before the operation itself. So what do I mean by that? Well, in this case, we need to evaluate n minus 1 before we pass the results to foo. And what we see in the LLVM IR is that we have an addition operation that computes n minus 1. And then the result of that-- stored into register 4-- gets passed to the call instruction on the next line, which calls out to function foo. Sound good?

Similarly, we need to evaluate n minus 2 before passing its results to the function bar. And we see that sequence of instructions showing up next in the LLVM IR. And now, we actually need the return value-- oh, yeah? Question?

**STUDENT:** What is NSW?

**TAO SCHARDL:** NSW? Essentially, that is an attribute, which we'll talk about later. These are things that decorate the instructions, as well as the types, within LLVM IR, basically, as the compiler figures stuff out. So it helps the compiler along with analysis and optimization. Good?

So for the last operation here, we had to evaluate both foo and bar and get their return values before we could add them together. And so the very last operation in this sequence is the addition. That just takes us those return values and computes their sum.

Now all of that used primitive types, in particular, integers. But it's possible that your code uses aggregate types. By aggregating types, I mean, arrays or struts, that sort of thing. And aggregate types are harder to store within registers, typically speaking. And so they're typically stored within memory.

As a result, if you want to access something within an aggregate type, if you want to read some elements out of an array, that involves performing a memory access or, more precisely, computing some address into memory, and then loading or storing that address. So here, for example, we have an array A of seven integers. And we're going to access A sub x.

In LLVM IR, that turns into two instructions-- this getelementptr followed by a load. And in the getelementptr case, this computes an address into memory and stores the result of that address into a register, in this case, register 5. The next instruction, the load, takes the address stored in register 5 and simply loads that particular memory address, storing the result into another register, in this case, 6. Pretty simple.

When reading the getelementptr instruction, the basic syntax involves a pointer into memory followed by a sequence of indices. And all that getelementptr really does is it computes an address by taking that pointer and then adding on that sequence of indices. So in this case, we have a getelementptr instruction, which takes the address in register 2, and then adds onto it-- yeah, that's a pointer into memory-- and then it adds onto it to indices.

One is the literal value 0, and the other is the value stored in register 4. So that just computes the address, starting at 2 plus 0 plus whatever was in register 4. That's all for straight line code. Good so far? feel free to interrupt if you have questions. Cool.

Functions-- let's talk about C functions. So when there's a function in your C code, generally speaking, you'll have a function within the LLVM code as well. And similarly, when there's a return statement in the C code, you'll end up with a return statement in the LLVM IR.

So here, we have just the bare bones C code for this fib routine. That corresponds to this fib function within LLVM IR. And the function declaration itself looks pretty similar to what you would get in ordinary C.

The return statement is also similar. It may take an argument, if you're returning some value to the caller. In this case, for the fib routine, we're going to return a 64-bit integer. And so we see that this return statement returns the 64-bit integer stored in register 0, a lot like in C.

Functions can have parameters. And when you have a C function with a list of parameters, basically, in LLVM IR, you're going to end up with a similar looking function with the exact same list of parameters translated into LLVM IR. So here, we have this C code for the mm base routine. And we have the corresponding LLVM IR for an mm-based function. And what we see is we have a pointer to a double as the first parameter, followed by a 32-bit integer, followed by another pointer to a double, followed by another 32-bit integer, following another pointer to a double, and another 33-bit integer, and another 32-bit integer.

One implicit thing with an LLVM IR if you're looking at a function declaration or definition, the parameters are automatically named %0, %1, %2, so on and so forth. There's one unfortunate thing about LLVM IR. The registers are a lot like C functions, but unfortunately, that implies that when you're reading LLVM IR, it's a lot like reading the code from your teammate, who always insists on naming things with nondescript, single-letter variable names. Also, that teammate doesn't comment his code, or her code, or their code.

OK, so basic blocks-- when we look at the code within a function, that code gets partitioned into chunks, which are called basic blocks. A basic block has a property that's a sequence of instructions. In other words, it's a blob a straight line code, where control can only enter from the first instruction in that block. And it can only leave from the last instruction in that block.

So here we have the C code for this routine fib.c. We're going to see a lot of this routine fib.c, by the way. And we have the corresponding LLVM IR. And what we have in the C code, what the C code is telling us is that if n is less than 2, you want to do one thing. Otherwise, you want to do some complicated computation and then return that result.

And if we think about that. We've got this branch in our control flow. And what we'll end up with are three different blocks within the LLVM IR. So we end up with one block, which does the computation is n less than 2. And then we end up with another block that says, well, in one case, just go ahead and return something, in this case, the input to the function. In the other case, do some complicated calculations, some straight line code, and then return that result.

Now when we partition the code of a function into these basic blocks, we actually have connections between the basic blocks based on how control can move between the basic blocks. These control flow instructions, in particular, the branch instructions, as we'll see, induce edges among these basic blocks. Whenever there's a branch instruction that can specify, that control can leave this basic block and go to that other basic block, or that other basic block, or maybe one or the other, depending on how the result of some computation unfolded.

And so for the fib function that we saw before, we had those three basic blocks. And based on whether or not n was than 2, either we would execute the simple return statement, or we would execute the blob of straight line code shown on the left. So those are basic blocks and functions. Everyone still good so far? Any questions? Clear as mud?

Let's talk about conditionals. You've already seen one of these conditionals. That's given rise to these basic blocks and these control flow edges. So let's tease that apart a little bit further. When we have a C conditional-- in other words, an if-then-else statement or a switch statement, for that matter-- that gets translated into a conditional branch instruction, or BR, in the LLVM IR representation.

So what we saw before is that we have this if n less than 2 and this basic block with two outgoing edges. If we take a really close look at that first basic block, we can tease it apart and see what each operation does. So first, in order to do this conditional operation, we need to compute whether or not n is less than 2. We need to do a comparison between n and the literal value 2.

That comparison operation turns into an icmp instruction within the LLVM IR, an integer comparison in the LLVM IR. The result of that comparison then gets passed to a conditional branch as one of its arguments, and the conditional branch specifies a couple of things beyond that one argument.

In particular, that conditional branch takes out 1-bit integer-- that Boolean result-- as well as labels of two different basic blocks. So that Boolean value is called the predicate. And that's, in this case, a result of that comparison from before. And then the two basic blocks say where to go if the predicate is true or where to go if the predicate is false. The first label is the destination when it's true, second label destination when it's false-- pretty straightforward.

And if we decide to map this onto our control flow graph, which we were looking at before, we can identify the two branches coming out of our first basic block as either the true branch or the false branch based on whether or not you follow that edge when the predicate is true or you follow it when the predicate is false. Sound good? That should be straightforward. Let me know if it's not. Let me know if it's confusing.

Now it's also possible that you can have an unconditional branch in LLVM IR. You can just have a branch instruction with one operand, and that one operand specifies a basic block. There's no predicate. There is no true or false. It's just the one basic block. And what that instruction says is, when you get here, now, go to that other basic block.

This might seem kind of silly, right? Why wouldn't we just need to jump to another basic block? Why not just merge this code with the code in the subsequent basic block? Any thoughts?

**STUDENT:** For instance, in this case, other things might jump in.

**TAO SCHARDL:** Correct. Other things might go to that basic block. And in general, when we look at the structure that we get for any particular conditional in C, we end up with this sort of diamond shape. And in order to implement that diamond shape, we need these unconditional branches. So there's a good reason for them to be around.

And here, we just have an example of a slightly more complicated conditional that creates this diamond shape in our control flow graph. So lets tease this piece of code apart. In the first block, we're going to evaluate if some predicate-- and in this case, our predicate is x bitwise and 1.

And what we see in the first basic block is that we compute the bitwise and store that result, do a comparison between that result, and the value 1. That gives us a Boolean value, which is stored in register 3. And we branch conditionally on whether 3 is true or false.

In the case that it's true, we'll branch to block 4. And in block 4, that contains the code for the consequence, the then clause of the if, then, else. And in the call square, we just call function foo. And then we need to leave the conditional, so we'll just branch unconditionally.

The alternative, if x and 1 is zero, if it's false, then we will execute the function bar, but then also need to leave the conditional. And so we see in block 5, following the false branch that we call bar, then we'd just branch to block 6. And finally, in block 6, we return the result.

So we end up with this diamond pattern whenever we have a conditional, in general. We may delete certain basic blocks if the conditional in the code is particularly simple. But in general, it's going to be this kind of diamond-looking thing. Everyone good so far?

One last C construct-- loops. Unfortunately, this is the most complicated C construct when it comes to the LLVM IR. But things haven't been too bad so far. So yeah, let's walk into this with some confidence.

So the simple part is that what we will see is the C code for a loop translates into LLVM IR that, in the control flow graph representation, is a loop. So a loop in C is literally a loop in this graph representation, which is kind of nice. But to figure out what's really going on with these loops, let's first tease apart the components of a C loop. Because we have a couple of different pieces in an arbitrary C loop.

We have a loop body, which is what's executed on each iteration. And then we have some loop control, which manages all of the iterations of that loop. So in this case, we have a simple C loop, which multiplies each element of an input vector x by some scale over a and stores the result into y.

That body gets translated into a blob of straight line code. I won't step through all of the straight line code just now. There's plenty of it, and you'll be able to see the slides after this lecture. But that blob of straight line code corresponds to a loop body. And the rest of the code in the LLVM IR snippet corresponds to the loop control.

So we have the initial assignment of the induction variable. The comparison would be end of the loop and the increment operation at the end. All of that gets encoded in the stuff highlighted in yellow, that loop control part.

Now if we take a look at this code, there's one odd piece that we haven't really understood yet, and it's this phi instruction at the beginning. The phi instruction is weird, and it arises pretty commonly when you're dealing with loops. It basically is there to solve a problem with LLVM's representation of the code. So before we describe the phi instruction, let's actually take a look at the problem that this phi instruction tries to solve.

So let's first tease apart the loop to reveal the problem. The C loop produces this looping pattern in the control flow graph, literally, an edge that goes back to the beginning. If we look at the different basic blocks we have, we have one block at the beginning, which initializes the induction variable and sees if there are any iterations of the loop that need to be run.

If there aren't any iterations, then they'll branch directly to the end of loop. It will just skip the loop entirely. No need to try to execute any of that code. And in this case, it will simply return.

And then inside the loop block, we have these two incoming edges-- one from the entry point of the loop, where i has just been set to zero, and another where we're repeating the loop, where we've decided there's one more iteration to execute. And we're going to go back from the end of the loop to the beginning. And that back edge is what creates the loop structure in the control flow graph. Make sense? I at least see one nod over there. So that's encouraging.

OK, so if we take a look at the loop control, there are a couple of components to that loop control. There's the initialization of the induction variable. There is the condition, and there's the increment. Condition says when do you exit. Increment updates the value of the induction variable.

And we can translate each of these components from the C code for the loop control into the LLVM IR code for that loop. So the increment, we would expect to see some sort of addition where we add 1 to some register somewhere. And lo and behold, there is an add operation. So we'll call that the increment.

For the condition, we expect some comparison operation and a conditional branch based on that comparison. Look at that. Right after the increment, there's a compare and a conditional branch that we'll either take us back to the beginning of the loop or out of the loop entirely.

And we do see that there is some form of initialization. The initial value of this induction variable is 0. And we do see a 0 among this loop control code. It's kind of squirreled away in that weird notation there. And that weird notation is sitting next to the phi instruction.

What's not so clear here is where exactly is the induction variable. We had this single variable i in our C code. And what we're looking at in the LLVM IR are a whole bunch of different registers. We have a register that stores what we're claiming to be i plus 1, then we do this comparison and branch thing. And then we have this phi instruction that takes 0 or the result of the increment.

Where did i actually go? So the problem here is that i is really represented across all of those instructions. And that happens because the value of the induction variable changes as you execute the loop. The value of i is different on iteration 0 versus iteration 1 versus iteration 2 versus iteration 3 and so on and so forth. i is changing as you execute the loop.

And there's this funny invariant. Yeah, so if we try to map that induction variable to the LLVM IR, it kind of maps to all of these locations. It maps to various uses in the loop body. It maps, roughly speaking, to the return value of this field instruction, even though we're not sure what that's all about. But we can tell it maps to that, because we're going to increment that later on. And we're going to use that in a comparison. So it kind of maps all over the place.

And because it changes values with the increment operation, we're going to encounter-- so why does it change registers? Well, we have this property in LLVM that each instruction defines the value of a register, at most, once. So for any particular register with LLVM, we can identify a unique place in the code of the function that defines that register value.

This invariant is called the static single assignment invariant. And it seems a little bit weird, but it turns out to be an extremely powerful invariant within the compiler. It assists with a lot of the compiler analysis. And it also can help with reading the LLVM IR if you expect it.

So this is a nice invariant, but it poses a problem when we're dealing with induction variables, which change as the loop unfolds. And so what happens when control flow merges at the entry point of a loop, for example? How do we define what the induction variable is at that location? Because it could either be 0, if this is the first time through the loop, or whatever you lost incremented. And the solution to that problem is the phi instruction.

The phi instruction defines a register that says, depending on how you get to this location in the code, this register will have one of several different values. And the phi instruction simply lists what the value of that register will be, depending on which basic block you came from.

So in this particular code, the phi instruction says, if you came from block 6, which was the entry point of the loop, where you initially checked if there were any loop iterations to perform, if you come from that block, then this register 9 is going to adopt the value 0. If, however, you followed the back edge of the loop, then the register is going to adopt the value, in this case, 14. And 14, lo and behold, is the result of the incremental operation. And so this phi instruction says, either you're going to start from zero, or you're going to be i plus 1.

Just to note, the phi instruction is not a real instruction. It's really a solution to a problem with an LLVM. And when you translate this code into assembly, the phi instruction isn't going to map to any particular assembly instruction. It's really a representational trick. Does that make some sense? Any questions about that? Yeah?

**STUDENT:** Why is it called phi?

**TAO SCHARDL:** Why is it called phi? That's a great question. I actually don't know why they chose the name phi. I don't think they had a particular affinity for the Golden Ratio, but I'm not sure what the rationale was. I don't know if anyone else knows. Yeah? Google knows all, sort of. Yeah, so adopt the value 0 from block 6 or 14 from block 8.

So that's all of the basic components of C translated into LLVM IR. The last thing I want to leave you with in this section on LLVM IR is a discussion of these attributes. And we already saw one of these attributes before. It was this NSW thing attached the add instruction.

In general, these LLVM IR constructs might be decorated with these extra words and keywords. And those are the keywords I'm referring to as attributes. Those attributes convey a variety of information. So in this case, what we have here is C code that performs this memory calculation, which you might have seen from our previous lecture. And what we see in the corresponding LLVM IR is that there's some extra stuff tacked onto that load instruction where you load memory.

One of those pieces of extra information is this align 4. And what that align 4 attribute says is it describes the alignment of that read from memory. And so if subsequent stages of the compiler can employ that information, if they can optimize reads that are 4-byte aligned, then this attribute will say, this is a load that you can go ahead and optimize.

There are a bunch of places where attributes might come from. Some of them are derived directly from the source code. If you write a function that takes a parameter marked as const, or marked as restrict, then in the LLVM IR, you might see that the corresponding function parameter is marked as no alias, because the restricted keyword said this pointer can ever alias or the const keyword says, you're only ever going to read from this pointer. So this pointer is going to be marked read-only. So in that case, the source code itself-- the C code-- was the source of the information for those attributes.

There are some other attributes that occur simply because the compiler is smart, and it does some clever analysis. So in this case, the LLVM IR has a load operation that's 8-byte aligned. It was really analysis that figured out the alignment of that load operation. Good so far? Cool.

So let's summarize this part of the discussion with what we've seen about LLVM IR. LLVM IR is similar to assembly, but a lot simpler in many, many ways. All of the computed values are stored in registers. And, really, when you're reading LLVM IR, you can think of those registers a lot like ordinary C variables.

LLVM IR is a little bit funny in that it adopts a static, single assignment paradigm-- this invariant-- where each registered name, each variable is written by, at most, one instruction within the LLVM IR code. So if you're ever curious where %14 is defined within this function, just do a search for where %14 is on the left-hand side of an equals, and there you go.

We can model of function in LLVM IR as a control flow graph, whose nodes correspond to basic blocks-- these blobs of straight line code-- and whose edges do node control flow among those basic blocks. And compared to C, LLVM IR is pretty similar, except that all of these operations are explicit. The types are explicit everywhere.

The integer sizes are all apparent. You don't have to remember that int really means a 32-bit integer, and you need n-64 to be a 64-bit integer, or you need a long or anything. It's just i and then a bit width.

There no implicit operations at the LLVM IR level. All the typecasts are explicit. In some sense, LLVM IR is like assembly if assembly were more like c. And that's doubly a statement that would not have made sense 40 minutes ago.

All right, so you've seen how to translate C code into LLVM IR. There's one last step. We want to translate the LLVM IR into assembly. And it turns out that structurally speaking, LLVM IR is very similar to assembly. We can, more or less, map each line of LLVM IR to some sequence of lines in the final assembly code.

But there is some additional complexity. The compiler isn't done with its work yet when it's compiling C to LLVM IR to assembly. There are three main tasks that the compiler still has to perform in order to generate x86-64. First, it has to select the actual x86 assembly instructions that are going to implement these various LLVM IR operations.

It has to decide which general purpose registers are going to hold different values and which values need to be squirreled away into memory, because it just has no other choice. And it has to coordinate all of the function calls. And it's not just the function calls within this particular source file. It's also function calls between that source file, and other source files that you're compiling, and binary libraries that are just sitting on the system. But the compiler never really gets to touch.

It has to coordinate all of those calls. That's a bit complicated. That is going to be the reason for a lot of the remaining complexity. And that's what brings our discussion to the Linux x86-64 calling convention. This isn't a very fun convention. Don't worry. But nevertheless, it's useful.

So to talk about this convention, let's first take a look at how a program gets laid out in memory when you run it. So when a program executes, virtually memory gets organized into a whole bunch of different chunks which are called segments. There's a segment that corresponds to the stack that's actually located near the top of virtual memory, and it grows downwards. The stack grows down. Remember this.

There is a heap segment, which grows upwards from a middle location in memory. And those two dynamically-allocated segments live at the top of the virtual address space. There are then two additional segments-- the bss segment for uninitialized data and the data segment for initialized data. And finally, at the bottom of virtual address space, there's a tech segment. And that just stores the code of the program itself.

Now when you read assembly code directly, you'll see that the assembly code contains more than just some labels and some instructions. In fact, it's decorated with a whole bunch of other stuff. And these are called assembler directives, and these directives operate on different sections of the assembly code.

Some of those directives refer to the various segments of virtual memory. And those segment directives are used to organize the content of the assembly file. For example, the .text directive identifies some chunk of the assembly, which is really code and should be located in the text segment when the program is run. The .bss segment identifies stuff that lives in the assembler directive to identify stuff in the bss segment. The .data directive identify stuff in the data segment, so on and so forth.

There are also various storage directives that will store content of some variety directly into the current segment-- whatever was last identified by a segment directive. So if, at some point, there is a directive x colon dot space 20, that space directive says, allocate some amount of memory. And in this case, it says, allocate 20 bytes of memory. And we're going to label that location x.

The .long segment says, store a constant long integer value-- in this case, 172-- in this example, at location y. The asciz segment similarly stores a string at that particular location. So here, we're storing the string 6.172 at location z. There is an align directive that aligns the next content in the assembly file to an 8-byte boundary.

There are additional segments for the linker to obey, and those are the scope and linkage directives. For example, you might see .globl in front of a label. And that single is linker that that particular symbol should be visible to the other files that the linker touches. In this case, .globl fib makes fib visible to the other object files, and that allows this other object files to call or refer to this fib location.

Now, let's turn our attention to the segment at the top, the stack segment. This segment is used to store data and memory in order to manage function calls and returns. That's a nice high-level description, but what exactly ends up in the stack segment? Why do we need a stack? What data will end up going there? Can anyone tell me?

**STUDENT:** Local variables in function?

**TAO SCHARDL:** Local variables in function. Anything else? You already answered once. I may call on you again. Go ahead.

**STUDENT:** Function arguments?

**TAO SCHARDL:** Sorry?

**STUDENT:** Function arguments?

**TAO SCHARDL:** Function arguments-- very good. Anything else? I thought I saw a hand over here, but--

**STUDENT:** The return address?

**TAO SCHARDL:** The return address. Anything else? Yeah? There's one other important thing that gets stored on stack. Yeah?

**STUDENT:** The return value?

**TAO SCHARDL:** The return value-- actually, that one's interesting. It might be stored on the stack, but it might not be stored on the stack. Good guess, though. Yeah?

**STUDENT:** Intermediate results?

**TAO SCHARDL:** Intermediate results, in a manner of speaking, yes. There are more intermediate results than meets the eye when it comes to assembly or comparing it to C. But in particular, by intermediate results, let's say, register state.

There are only so many registers on the machine. And sometimes, that's not enough. And so the function may want to squirrel away some data that's in registers and stash it somewhere in order to read it back later. The stack is a very natural place to do it. That's the dedicated place to do it. So yeah, that's pretty much all the content of what ends up on the call stack as the program executes.

Now, here's the thing. There are a whole bunch of functions in the program. Some of them may have been defined in the source file that you're compiling right now. Some of them might be defined in other source files. Some of them might be defined in libraries that were compiled by someone else, possibly using a different compiler, with different flags, under different parameters, presumably, for this architecture-- at least, one hopes. But those libraries are completely out of your control.

And now, we have this problem. All those object files might define these functions. And those functions want to call each other, regardless of where those functions are necessarily defined. And so somehow, we need to coordinate all those function calls and make sure that if one function wants to use these registers, and this other function wants to use the same registers, those functions aren't going to interfere with each other. Or if they both want to read stack memory, they're not going to clobber each other's stacks.

So how do we deal with this coordination problem? At a high level, what's the high-level strategy we're going to adopt to deal with this coordination problem?

**STUDENT:** Put the values of the registers on the stack before you go into the function.

**TAO SCHARDL:** That will be part of it. But for the higher level strategy-- so that's a component of this higher level strategy. Yeah? Go ahead.

**STUDENT:** Calling convention?

**TAO SCHARDL:** Calling convention. You remembered the title of this section of the talk. Great. We're going to make sure that every single function, regardless of where it's defined, they all abide by the same calling convention. So it's a standard that all the functions will obey in order to make sure they all play nicely together.

So let's unpack the Linux x86-64 calling convention. Well, not the whole thing, because it's actually pretty complicated, but at least enough to understand the basics of what's going on. So a high level, this calling convention organizes the stack segment into frames, such that each function instantiation-- each time you call a function-- that instantiation gets a single frame all to itself.

And to manage all those stack frames, the calling convention is going to use these two pointers-- rbp and rsp, which you should've seen last time. rbp, the base pointer, will point to the top of the current stack frame. rsp will point to the bottom up the current stack frame. And remember, the stack grows.

Now when the code executes call-and-return instructions, those instructions are going to operate on the stack, these various stock pointers, as well as the instruction pointer, rip, in order to manage the return address of each function.

In particular, when a call instruction gets executed, in x86, that call instruction will push the current value of rip onto the stack, and that will be the return address. And then the call instruction will jump to its operand. It's operand being the address of some function in the program memory, or, at least, one hopes. Perhaps there was buffer overflow corruption of some kind, and your program is in dire straits. But presumably, it's the address of a function.

The return instruction complements the call, and it's going to undo the operations of that call instruction. It'll pop the return address off the stack and put that into rip. And that will cause the execution to return to the caller and resume execution from the statement right after the original call. So that's the high level of how the stack gets managed as well as the return address.

How about, how do we maintain registers across all those calls? Well, there's a bit of a problem. Because we might have two different functions that want to use the same registers. Some of this might be review, by the way, from 6004. If you have questions, just let me know.

So we have this problem, where two different functions, function A, which might call another function B. Those two functions might want to use the same registers. So who's responsible for making sure that if function B operates on the same registers as A, that when B is done, A doesn't end up with corrupted state in its registers?

Well, they're two different strategies that could be adopted. One is to have the caller save off the register state before invoking a call. But that has some downsides. The caller might waste work, saying, well, I have to save all of this register state in case the function I'm calling wants to use those registers. If the calling function doesn't use those registers, that was a bunch of wasted work.

So on the other side, you might say, well, let's just have the callee save all that registered state. But that could waste work if the callee is going to save off register state that the caller wasn't using. So if the callee says, well, I want to use all these registers. I don't know what the calling function used, so I'm just going to push everything on the stack, that could be a lot of wasted work.

So what does the x86 calling convention do, if you had to guess? Yeah?

**STUDENT:** [INAUDIBLE]

**TAO SCHARDL:** That's exactly right. It does a little bit of both. It specifies some of the registers as being callee-saved registers, and the rest of the registers are caller-saved registers. And so the caller will be responsible for saving some stuff. The callee will be responsible for saving other stuff. And if either of those functions doesn't need one of those registers, then it can avoid wasted work.

In x86-64, in this calling convention, turns out that the rbx, rbp, and r12 through r15 registers are all callee saved, and the rest of the registers are caller saved. In particular, the C linkage defined by this calling convention for all the registers looks something like this. And that identifies lots of stuff. It identifies a register for storing the return value, registers for storing a bunch of the arguments, caller-save registers, callee-saved registers, a register just for linking.

I don't expect you to memorize this in 12 seconds. And I think on any quiz-- well, I won't say what the course app will do on quizzes this year.

**STUDENT:** [INAUDIBLE] everyone.

**TAO SCHARDL:** Yeah, OK, well, there you go. So you'll have these slides later. You can practice memorizing them. Not sure on this slide. There are a couple other registers that are used for saving function arguments and return values. And, in particular, whenever you're passing floating point stuff around, the xmm register 0 through 7 are used to deal with those floating point values.

Cool. So we have strategies for maintaining the stack. We have strategies for maintaining register states. But we still have the situation where functions may want to use overlapping parts of stack memory. And so we need to coordinate how all those functions are going to use the stack memory itself.

This is a bit hard to describe. The cleanest way I know describe it is just to work through an example. So here's the setup. Let's imagine that we have some function A that is called of function B. And we're in the midst of executing function B, and now, function B is about to call some other function C.

As we mentioned before, B has a frame all to itself. And that frame contains a whole bunch of stuff. It contains arguments that A passed to B. It contains a return address. It contains a base pointer. It contains some local variables. And because B is about to call C, it's also going to contain some data for arguments that B will pass to C.

So that's our setup. We have one function ready to call another. Let's take a look at how this stack memory is organized first. So at the top, we have what's called a linkage block. And in this linkage block, this is the region of stack memory, where function B will access non-register arguments from its caller, function A.

It will access these by indexing off of the base pointer, rbp, using positive offsets. Again, the stack grows down. B will also have a block of stack space after the linkage block and return address and bass pointer. It will have a region of its frame for local variables, and it can access those local variables by indexing off of rbp in the negative direction. Stack grows down. If you don't have anything else, stack grows down.

Now B is about to call a function C, and we want to see how all of this unfolds. So before calling C, B is going to place non-register arguments for C on to a reserved linkage block in its own stack memory below its local variables. And it will access those by indexing rbp with negative offsets.

So those arguments from B to its callers will specify those to be arguments from B to C. And then what's going to happen? Then B is going to call C. And as we saw before, the call instruction saves off the return address onto the stack, and then it branches control to the entry point of function C.

When the function C starts, it's going to execute what's called the function prologue. And the function prologue consists of a couple of steps. First, it's going to save off the base pointer for B's stack frame. So it'll just squirrel away the value of rbp onto the stack. Then it's going to set rbp equal to rsp, because we're now entering a brand new frame for the invocation of C.

And then C can go ahead and allocate the space that it needs on the stack. This will be space that C needs for its own local variables, as well as space that C will use for any linkage blocks that it creates for the things that it calls.

Now there is one common optimization that the compiler will attempt to perform. If a function never needs to perform stack allocations, except to handle these function calls-- in other words, if the difference between rbp and rsp is a compile time constant, then the compiler might go ahead and just get rid of rbp and do all of the indexing based off the stack pointer rsp.

And the reason it'll do that is because, if it could get one more general purpose register out of our rbp, well, now, rpb is general purpose. And it has one extra register to use to do all of its calculations. Reading from a register takes some time. Reading from even L1 cache takes significantly more, I think, four times that amount. And so this is a common optimization that the compiler will want to perform.

Now, turns out that there's a lot more to the calling convention than just what's shown on these slides. We're not going to go through that today. If you'd like to have more details, there's a nice document-- the *System* V *ABI*-- that describes the whole calling convention. Any questions so far?

All right, so let's wrap all this up with a final case study, and let's take a look at how all these components fit together. When we're translating a simple C function to compute Fibonacci numbers all the way down to assembly. And as you've been describing this whole time, we're going to take this in two steps.

Let's describe our starting point, fib.c. This should be basically no surprise to you at this point. This is a C function fib, which computes the nth Fibonacci number in one of the worst computational ways possible, it turns out. But it computes the nth Fibonacci number f of n recursively using the formula f of n is equal to n when n is either 0 or 1. Or it computes f of n minus 1 and f of n minus 2 and takes their sum.

This is an exponential time algorithm to compute Fibonacci numbers. I would say, don't run this at home, except, invariably, you'll run this at home. There are much faster algorithms to compute Fibonacci numbers. But this is good enough for a didactic example. We're not really worried about how fast can we compute fib today.

Now the C code fib.c is even simpler than the recurrence implies. We're not even going to bother checking that the input value n is some non-negative value. What we're going to do is say, look, if n is less than 2, go ahead and return that value of n. Otherwise, do the recursive thing.

We've already seen this go a couple of times. Everyone good so far? Any questions on these three lines? Great.

All right, so let's translate fib.c into fib.ll. We've seen a lot of these pieces in lectures so far. And here, we've just rewritten fib.c a little bit to make drawing all the lines a little bit simpler. So here, we have the C code for fib.c.

The corresponding LLVM IR looks like this. And as we could guess from looking at the code for fib.c, we have this conditional and then two different things that might occur based on whether or not n is less than 2. And so we end up with three basic blocks within the LLVM IR.

The first basic block checks event is less than 2 and then branches based on that result. And we've seen how all that works previously. If n happens to be less than 2, then the consequent-- the true case of that branch-- ends up showing up at the end. And all it does is it returns the input value, which is stored in register 0.

Otherwise, it's going to do some straight line code to compute fib of n minus 1 and fib of n minus 2. It will take those return values, add them together, return that result. That's the end Fibonacci number. So that gets us from C code to LLVM IR. Questions about that? All right, fib n minus 1, fib n minus 2, add them, return it. We're good.

OK, so one last step. We want to compile LLVM IR all the way down to assembly. As I alluded to before, roughly speaking, the structure of the LLVM IR resembles the structure of the assembly code. There's just extra stuff in the assembly code. And so we're going to translate the LLVM IR, more or less, line by line into the assembly code and see where that extra stuff shows up.

So at the beginning, we have a function. We were defining a function fib. And in the assembly code, we make sure that fib is a globally accessible function using some assembler directives, the globlfib directive. We do an alignment to make sure that function lies in a nice location in the instruction memory, and then we declare the symbol fib, which just defines where this function lives in memory.

All right, let's take a look at this assembly. The next thing that we see here are these two instructions-- a push queue or rbp and a movq of rsp, rbp. Who can tell me what these do? Yes?

**STUDENT:** Push the base [INAUDIBLE] on the stack, then [INAUDIBLE].

**TAO SCHARDL:** Cool. Does that sound like a familiar thing we described earlier in this lecture?

**STUDENT:** the calling convention?

**TAO SCHARDL:** Yep, it's part of the calling convention. This is part of the function prologue. Save off rpb, and then set rbp equal to rsp. So we already have a couple extra instructions that weren't in the LLVM IR, but must be in the assembly in order to coordinate everyone.

OK, so now, we have these two instructions. We're now going to push a couple more registers onto the stack. So why does the assembly do this? Any guesses? Yeah?

**STUDENT:** Callee-saved registers?

**TAO SCHARDL:** Callee-saved registers-- yes, callee-saved registers. The fib routing, we're guessing, will want to use r14 rbx during this calculation. And so if there are interesting values in those registers, save them off onto the stack. Presumably, we'll restore them later.

Then we have this move instruction for rdi into rbx. This requires a little bit more arcane knowledge, but any guesses as to what this is for?

**STUDENT:** rdi is probably the argument to the function.

**TAO SCHARDL:** rdi is the argument to the function. Exactly. That's the arcane knowledge. So this is implicit from the assembly, which is why you either have to memorize that huge chart of GPR C linkage nonsense. But all this operation does is it takes whatever that argument was, and it's squirrels it away into the rbx register for some purpose that we'll find out about soon.

Then we have this instruction, and this corresponds to the highlighted instruction on the left, in case that gives any hints. What does this instruction do?

**STUDENT:** [INAUDIBLE].

**TAO SCHARDL:** Sorry.

**STUDENT:** It calculates whether n is small [INAUDIBLE].

**TAO SCHARDL:** Correct. It evaluates the predicate. It's just going to do a comparison between the value of n and the literal value of 2, comparing against 2.

So based on the result of that comparison, if you recall, last lecture, the results of a comparison will set some bits in this implicit EFLAGS flags register, or RFLAGS register. And based on the setting of those bits, the various conditional jumps that occur next in the code will have varying behavior. So in case the comparison results to false-- if n is, in fact, greater than or equal to 2-- then the next instruction is jge, will jump to the label LBB0 underscore 1. You can tell already that reading assembly is super-fun.

Now that's a conditional jump. And it's possible that the setting of bits in RFLAGS doesn't evaluate true for that condition code. And so it's possible that the code will just fall through pass this jge instruction and, instead, execute these operations. And these operations correspond to the true side of the LLVM IR branch operation. When n is less than 2, this will move n into rax, and then jumped to the label LBB03. Any guesses as to why it moves n into our rax? Yeah?

**STUDENT:** That's the return value.

**TAO SCHARDL:** That's a return value-- exactly. If it can return a value through registers, it will return it through rax. Very good. So now, we see this label LBBO1. That's the label, as we saw before, for the false side of the LLVM branch.

And the first thing in that label is this operation-- leaq minus 1 of rbx rdi. Any guesses as to what that's for? The corresponding LLVM IR is highlighted on the left, by the way.

The lea instruction means load-effective address. All lea does is an address calculation. But something that compilers really like to do is exploit the lea instruction to do simple integer arithmetic as long as that integer arithmetic fits with the things that lea can actually compute.

And so all this instruction is doing is adding negative 1 to rbx. And rbx, as we recall, stored the input value of n. And it will store the result into rdi. That's all that this instruction does. So it computes the negative 1, stores it into rbi.

How about this instruction? This one should be easier.

**STUDENT:** For the previous one, how did you get [INAUDIBLE]? I'm familiar with [INAUDIBLE] because [INAUDIBLE]. But is there no add immediate instruction in x86?

**TAO SCHARDL:** Is there no add immediate instruction? So you can do an add instruction in x86 and specify an immediate value. The advantage of this instruction is that you can specify a different destination operand. That's why compilers like to use it. More arcane knowledge. I don't blame you if this kind of thing turns you off from reading x86. It certainly turns me off from reading x86.

So this instruction should be a little bit easier. Guess as to why it does? Feel free to shout it out, because we're running a little short on time.

**STUDENT:** Calls a function.

**TAO SCHARDL:** Calls a function. What function?

**STUDENT:** Call fib.

**TAO SCHARDL:** Call fib, exactly. Great. Then we have this move operation, which moves rax into r14. Any guess as to why we do this? Say it.

**STUDENT:** Get the result of the call.

**TAO SCHARDL:** Get the result of the call. So rax is going to store the return value of that call. And we're just going to squirrel it away into r14. Question?

**STUDENT:** [INAUDIBLE]

**TAO SCHARDL:** Sorry.

**STUDENT:** It stores [INAUDIBLE]?

**TAO SCHARDL:** It'll actually store the whole return value from the previous call.

**STUDENT:** [INAUDIBLE]

**TAO SCHARDL:** It's part of that result. This will be a component in computing the return value for this call of fib. You're exactly right. But we need to save off this result, because we're going to do, as we see, another call to fib. And that's going to clobber rax. Make sense? Cool.

So rax stores the result of the function. Save it into r14. Great. Since we're running short of time, anyone want to tell me really quickly what these instructions do? Just a wild guess if you had to.

**STUDENT:** N minus 2

**TAO SCHARDL:** n minus 2. Compute n minus 2 by this addition operation. Stash it into rdi. And then you call fib on n minus 2. And that will return the results into rax, as we saw before. So now, we do this operation. Add r14 into rax. And this does what?

**STUDENT:** Ends our last function return to what was going off this one.

**TAO SCHARDL:** Exactly. So rax stores the result of the last function return. Add it into r14, which is where we stashed the result of fib of n minus 1. Cool.

Then we have a label for the true side of the branch. This is the last pop quiz question I'll ask. Pop quiz-- God, I didn't even intend that one. Why do we do these pop operations? In the front.

**STUDENT:** To restore the register before exiting the stack frame?

**TAO SCHARDL:** Restore the registers before exiting the stack frame-- exactly. In calling convention terms, that's called the function epilogue. And then finally, we return.

So that is how we get from C to assembly. This is just a summary slide of everything we covered today. We took the trip from C to assembly via LLVM IR. And we saw how we can represent things in a control flow graph as basic blocks connected by control flow edges. And then there's additional complexity when you get to the actual assembly, mostly to deal with this calling invention.

That's all I have for you today. Thanks for your time.