CHARLES LEISERSON: So today we're going to take a little bit closer look at what's happening under the covers when you compile a C program. But before we get into that, we did a little interesting correlation on your scores for the first problem, for the every bit one. So this is basically plotting. It's a scatter plot of how you did in your test coverage score versus how you did in your correctness score and performance, correctness and performance together.

And what's interesting is that if you did better in your test coverage, you did better in your performance and correctness. OK, that's a pretty good correlation, right? There's some outliers here. But that's a pretty good correlation. Yeah, John? Yeah?

JOHN: The--

CHARLES LEISERSON: Do we have a handheld? Here we go. Just a second. The only thing is we have to figure out how to turn it on. There we go.

JOHN: Yeah, so just to clarify, the people who seemingly got no test coverage but really good performance scores, what actually happened was that we tested for things that we didn't expect students to cover for like on feeding invalid values to better a set and better a get or testing for private functions to their implementations. So in reality they had better test suites than the coverage score would indicate.

CHARLES LEISERSON: So what are the lessons that one draws from this? So professional engineers know what the lessons are. So the lessons are that it is actually better, if you have a coding problem to do, to write tests first. Before you code you write your tests.

And that actually speeds the development of fast correct code. It's actually faster. You get to the end result much faster. Because whenever you make an error in your

program, you instantly know that you may have a problem rather than thinking that you're doing something OK and then discovering that, oh, in fact your code is, in fact, incorrect, and you're working away optimizing something that's not working So before coding, it's highly recommended that you write test.

Also if you find a bug, when you find a bug, the first thing you should do is write a test for that bug if it wasn't already covered. Then you fix the bug. And then you make sure that your test now, that your new implementation, passes that particular one. Professional engineers know this. Professional software developers know this. It comes hard. And if you want a job at any top flight software firm, they're going to expect that you know that you write tests first before you do coding.

The second lesson isn't quite so obvious from this. But it's the second lesson that I think some people experienced in the class which was the idea of putting you in groups, in particular in pairs, was not so that you could do divide and conquer on the code. It was to do pair programming.

And what we found was that a bunch of groups divided up the work. And they said, OK, it'll go faster if you do this one and I do that one. Once again that's probably a mistake. If you can sit together and take turns at the keyboard making the changes, it may seem like it's going slower to begin with, but it's amazing how many errors you catch and how quickly you find your errors because you're just talking with each other. And it's like, oh, duh.

So good programmers know this. That it really helps to have more than one person understand what's going on in the code. So the people who had difficulty with their partners one way or another often did not, it was partly because they just divided up the work, you're responsible for that, oh, we got a bad grade on that, that's your fault. No, both partners own that grade 100%. And the best way to ensure is to work together.

Now, this sometimes flies in the face of people who believe that they are clever or more experienced than somebody, than their partner, oh, I can do this much better on my own. Usually, that's true for little projects, but as the projects get bigger that

becomes a much harder situation to deal with. It becomes the case that you really want two brains looking at the same thing, four eyes as opposed to two eyes looking at things.

But I think, in particular, before coding write test. And we are right now working on improving the infrastructure. One of the things that they have in most companies is, at the very minimum, they have what's called a nightly build. Nightly build says they take all the software, they build it, and then they run regression tests against it all night while everybody's home sleeping. Come in the next morning, here's the things that broke. And if you broke the build, you got some work to do that morning. And it's generally not a good idea to break the build.

What has been demonstrating, in fact, is that continuous build is even better. This is where, whenever you make a change to the program, you run the full suite of tests on it. And we're going to look into it. We have to see what our resources are. As you know, our TAs are a limited resource.

But we're going to look into seeing whether we can provide more of that kind of infrastructure on some of the later projects for you folks. So you can sort of see the matrix that we eventually got to you. You can see that develop in real time. How am I doing against other people's tests? How are they doing against my tests, et cetera? So we'll see whether we can do that.

But what's funny is you think that it'd be faster to just code and do it. Computer science is full of wonderful paradoxes. And one of them is that doing things like writing the extra code to test is actually faster than not writing it, surprisingly. It really gets you to the end result a lot faster. Any questions about that? Any comments about that?

Let's talk about our today. So today we're going to talk mostly about single threaded performance. This is one instruction stream that you're trying to make go fast. But if you look at today's computing milieu, how all of the computers are used, what do you have?

You've got networks of multi-core clusters. It's parallelism everywhere. You've got shared memory among processors within a chip. You've got message passing among machines in a cluster. You've got network protocols among clusters so that you can do wide area things. Yet we're saying, no, let's take a look at what happens on one core on one machine.

So why is that important to focus first on what one core can do? Why study single threaded performance at all? Let's just go do the parallel stuff. That's more fun anyway.

Well, there are a couple of reasons that I can think of. The first one is at the end of the day, even if you've got something running widely in parallel, the code is running in each core in a single threaded manner. You just have a bunch of them. And so if you've given up a factor of two or a factor of four in performance or even more, as you're aware, you can sometimes make it orders of magnitude, but in performance what you're saying is that you're going to end up using much more resources to do your particular job in parallel. And resources is money.

So if I can do the job with a cluster of 16 processors and somebody else can do it in a cluster with only four processors, hey, they just spent a quarter the amount on not just the capital investment in that hardware but also the operating costs of what it cost to actually cool and provide electricity to and maintain and so forth. All that gets much cheaper. So if you get good single thread performance, it translates. That's kind of the direct reason.

The indirect reason is, for studying it, is that many of the lessons will generalize. So things that we'll see for single core, there is an analogy when you start looking at parallel and distributed systems. So that's a little less concrete. But as you'll see as you gain experience, you'll see that there's a lot of lessons that generalize to how do you think about performance no matter what the context.

So what about a single threaded machine? What's it like? So some of this is going to be a little bit review, but we're going to just sort of go deeper. We've sort of been taking layers off the onion. And today we're going to take a few more layers off the

onion.

So you have inside a processor core. You've got registers. You've got the functional units to do your ALU operations, floating point units, vector units these days, and all the stuff to do instruction, execution, and coordination, scheduling, out of order execution, and so forth. In addition then, you have a memory hierarchy. Within the core, you typically have registers and L1 and L2 caches. And then outside the core often is the L3 cache. DRAM memory, you may have a solid-state drive these days and disk. And so in that context, you're trying to make your code run fast.

So when you compile the piece of code, so here I have a piece of code. I'm always amused when I put up a Fibonacci as the example. Because this is a really terrible way to compute Fibonacci numbers. So this is an exponential time algorithm for computing Fibonacci numbers. And you may be aware you can do this in linear time just by adding up from the bottom. In fact, if you take the algorithms course, you learn that you can actually do this in logarithmic time by matrix, recursive squaring of matrices.

So it's sort of interesting to put up something where we say we're going to optimize this. And, of course, we'll get a constant factor improvement on something like this. But, in fact, really this is a terrible program to write for optimization. But it's good didactically. And Fibonacci numbers are fun anyway.

So typically what happens is when you run GCC on your .C file and produce a binary, what happens is it produces the machine code, which is basically a string of bytes, zeros and ones. And that goes, when you run the program, that goes through the hardware interpreter. So the hardware of the machine is doing an interpretation of these very simple instructions and produces an execution.

But, in fact, there's actually four stages that go on inside of GCC if you type a command like this. The first thing is what's called preprocessing. And what that does is it does any macro expansion and so forth, things that are just basically on the level of textual substitutions before you get into the guts of the compiler. Then you actually do the compiler. And that produces a version of machine code called

assembly language, which we'll see in just a minute. And from that version of assembly language it then goes into a process called linking and loading, which actually causes it to produce the binary that you can then execute.

So all four stages are included here. And there are switches to GCC that let you do only one or all of these things. You can, for example, run the preprocessor GCC. You can tell it to run the preprocessor alone and see what all your macros expanded to. Yes? Question?

AUDIENCE: What's the difference between compiling and assembling?

CHARLES LEISERSON: So compiling reduces it to essentially assembly language. And then assembling is taking that assembly language and producing the machine binary.

AUDIENCE: I was going to say, there's a one-to-one correspondence between machine code and assembly, but there's not a one-to-one correspondence between C code and assembly.

CHARLES LEISERSON: Yeah. So there's actually not quite a one to one, but it's very close. It's very close. So you can think of it as one to one between assembly machine code. But assembly is, in some sense, a more human readable and understandable version of machine code. In fact, that's what we're going to talk about.

So let's go directly to assembly code. To do that I can use the minus S switch. Now it turns out it's also helpful to use the minus G switch. Minus G says give me all the debugger symbol tables. And what that makes it is so that you can actually read the assembly language.

If you don't have that information, then you don't know what the programmer wrote as the symbols. Instead you will get computer generated symbol names for things. And you don't have any meaning to those. So it's really a good idea to use minus G and minus S together.

And this basically provides a convenient symbolic representation of the machine language. And this is sort of the type of thing that you'll get, something coming out

that looks like this. It's basically an Ascii. It's in text, characters, rather than being in the binary executable.

And if you want, you can find out all the vagaries of it. This is one site that has some reasonable documentation on the GNU assembler. It's actually not as good on the instructions, but it's really good on all the directives, which we'll talk about in a minute like .global and .type and all that stuff. It's very good on that stuff.

There's another thing that you can do. And once again, it's also helpful if you've produced a binary that has the symbol table. And that is to do a dump of the object code. And when you do a dump of the object code, what it does is you basically give it an executable and it goes backwards the other way, take this executable and undo one step, disassemble it.

And what's good about object dump is that it gives you, first of all, these are all the byte codes of the instructions. Also if you've got the minus S says interleave the source code, so you can see, here's the source code interleaved. So you can see which regions of code depend on which things. And so it basically tells you where in memory it's being loaded, it's been loaded, what the instructions are. And then it gives you the assembly interpretation of that machine binary.

And this is where you can see it's almost one to one what's going on here. Here we have a push of an operand. And that notice is just a one byte code. Whereas here we've got an opcode and two arguments. And it has three bytes as it turns out. So you can see there's sort of a correspondence. Yeah, question?

AUDIENCE: How does [? logic then take the ?] machine language code and go to-- how does it know the function names and stuff?

CHARLES LEISERSON: It knows the function names because when you compile it with -g, it produces, in addition to producing the binary, it produces a separate segment that's not loaded in that has all that information that says, oh, at this location is where this symbol is. And it produces all that as stuff that's never loaded in at run time but which is there in order to aid debuggers mainly. Question?

AUDIENCE: To compile something not using the gflag and then you do an object dump, how would that work?

CHARLES LEISERSON: Then what happens is, first of all, you would not be able to get this stuff interleaved. And then things like here where it says fib, well, fib may be an external name so you might know it anyway. But if it were an internal name, you would not be able to see what it was. Yeah, if you're going to respond let's get you on mike here. Why don't you just hold this?

JOHN: Yeah, so you'll generally get the function names so you know roughly a huge blob of assembly corresponds to a function. But you won't be able to get any information about what variables are in which registers or what position the sixth line of assembly corresponds to in terms of your source code.

CHARLES LEISERSON: Then the other thing that you can do is you can actually take the assembler, the assembly code, if you produce just the assembly code, and if you tell GCC to take a .s file, which is the assembly code, you can produce the machine code from it. And so one thing that you can do is you can produce a .s file and then edit it in Emacs or VI or whatever your favorite text editor is and then assemble it with GCC. So you can actually modify what the machine code is directly. And that's what we're going to spend a little bit of time doing today. Let's go in and see what the compiler generates and then let's twiddle it a bit.

So here's what we're going to expect that you do, that you're able to do. We expect in this class that you're going to be able to understand how a compiler implements the C linguistic constructs using x86 instructions. We're going to expect that you can read x86 assembly language with the aid of a manual. We don't expect that you know all the instructions, but the basic ones we expect that you know what those are.

We expect that you're going to be able to make simple modifications to the assembly language generated by a compiler, and that you would know, if push came to shove, how to write your own machine code on your own. That's not something we're going to expect that you do, but you would know how to get started

to do that if at some point you said, oh, I really have to write this in assembler. So this is, as I say, really we're going to take off some layers of the onion today, try to get down what's going on.

It turns out this is actually kind of fun. Now, the part that's not fun at some level is the x86 64 machine model. The x86 is what's called a complex instruction set computer. And these, long ago, were demonstrated to be inferior to so-called reduced instruction set computers.

But that hasn't mattered in the marketplace. What's mattered in the marketplace is who could build better and faster chips. And also the amount of people who started using the x86 instruction set has produced a huge legacy and inertia.

It's sort of like some people might argue that Esperanto is a better language for everybody to learn than English. But how come English with all its complexities and so forth, and I'm sure for some of you who have learned English as a second language, it's like it's a crazy language. Who do you learn English? Well, it's because that's what everybody's learning. That's where the legacy is. And so x86 is very much like the English of machines these days.

So in this model there's basically a flat 64-bit address space. There are 16 64-bit general purpose registers, and then what are some segment registers, a register full of flags, an instruction pointer register, rest in peace. They're eight 80-bit floating point data registers, some control status registers, an opcode register, a floating point instruction pointer register, and a floating point data pointing register, some MMX registers for the multimedia extensions, and a 128-bit XMM registers for the SSE instructions, which are the ability to have an opcode run over several pieces of data at once, short vectors, vector instructions, and a 32-bit register that frankly I don't have a clue as to what it does.

So, fortunately, we don't have to know all these. You can look at the architecture manual if any of these become important. So our goal is not to memorize the x86 instruction set. That would be a punishment probably worse than death. The only thing worse would be learning all of C++.

So here's the general registers. So there are basically 64-bit registers. And here's the mnemonics that they have. So you can see is all very mnemonic, right? We got some of them that are numbered. How come they're all just not numbered? I mean come on, right? I know why. I know why. Don't tell me.

So what you get to do is look at and remember that there are all these fun registers. And what they did is the x86 64 architecture grew out of the x86 which was 32-bit. Well, in fact, originally it was 16-bit. And it's been extended twice to have more bits in the instruction word so that now it's a 64-bit instruction word.

And what they did in order to make it so that they could run legacy code more easily, which might have been written with a smaller word size, is they've overlap so that the EAX register, for example, is the low order 32-bits of the RAX register. So what you do is you'll see that R is the prefix that says, hey, that's a 64-bit register. E is the prefix that says that it is--

Whoops, I made a mistake there. Those should all be Es. Oh, no, sorry, no, there's are correct. These are R and then with D because these are the extended ones, yes. So these are D. So R and D, that means also that it's 16. So you can see just how easy this is to remember without a cheat sheet, right?

And then you go down to 15, et cetera. And so you can go all the way down to byte naming, the low order byte of the registers. In addition, it turns out that that's not all. But the high order byte of the 16-bit registers are also available as independently named registers.

When you're using this in a C program, there's a convention that C has. And it's actually different on Windows from on Linux. Because there's no reason they should make those things compatible. That would be too easy. So instead they have different ones. But the ones on Linux, this is essentially the structure. What happens when you call a subroutine is generally you're passing the arguments to the subroutine in registers. And in fact the first six arguments are passed in these registers.

RDI, you'll get very familiar with RDI. Because that's where the first argument is always passed. And almost all your functions will have a first argument, except for the ones that have no arguments, and then the second arguments, the third, and so forth, and then fifth and sixth. If you get more than six, then it turns out, then you start passing arguments through memory. But otherwise the convention is that the arguments are passed through registers.

There are a couple of other important registers. One here is the return value always comes back in RAX. So when a function returns, boom, that's where, RAX is where the value of the return is. There is a base pointer and a stack pointer which give you the stack frame so that when you do a push and want to push local variables those are telling you the limits of your local variats. And we'll talk more about that a little bit.

And then there are a variety of other ones. Some are callee saved and some are caller saved. And you can refer to this chart. And there are others similar to it in the various manuals.

Now, it gets pretty confusing, if this isn't confusing enough for the naming. Let's go on to how you name data types. And I think some of you have already experienced this a little bit, the beauties of the data types. So in C, they have all these different data types such as I'm listing here.

And if you want to generate a constant of that size, so sometimes the compiler will coerce a value from one type to another. But sometimes it won't. And so if you want to have a constant, and I've just given a couple things here, for example, if you want it to be just an int, you can just write the number. But if you want it to be unsigned, you have to put a U after it. Or if you want it to be a long, you have to put an L after it.

And for many things it'll get coerced automatically to the right type because if you do an operator with another argument it will be coerced to that type. But some of you got burned on some of the shift things to begin with because it wasn't clear what

exactly the sizes. Well, you can be explicit in C and name them using this particular convention. This tells you how many bytes are being allocated for that type in the x86 64 size. So it's [? veted ?] here for four.

Now, long double is a funny one. It's actually allocate 16 bytes, but only 10 of them are used. So basically there are six bytes that get unused by that. And I think that's for future expansion so that they can have even wider extension. This is generally used, of course, for floating point and so forth.

Now, in the assembly language, each of the operators has a suffix. And sometimes, if it's a two operand instruction, it may, where it's taking things of different sizes, it may have two suffixes. But it has a suffix which is a single character that tells you what the size is that you're working with.

So, for example, B is for byte. W is for word because originally the words were 16 bits. L is for long except that it's not a long so don't get confused. L is not long. Long is a quad word, or Q, four bytes. And then a float is an S. A double is a D. And a long double is a T.

So these you will get familiar with. And they're not so hard. But that doesn't mean you know them right off the bat. And it helps to have a cheat sheet. As I say, the main one not to get confused about is the Ls. L means something different in x86 than it means in C.

So, for example, here we have a move of, and because it's a Q, I know that it is an eight byte or a 64-bit operator. And you can tell that also because it's using RBP and RAX, both of which are 64-bit registers. In fact, in assembly, you can actually write it without the Q, because the assembler can infer when the Q isn't there that, oh, this is a 64-bit register, that's a 64-bit register, I bet he means move 64-bits. So it actually fills that in sometimes. But sometimes you need to be explicit. Question?

AUDIENCE: What happens when you actually put 64-bit registers but you only, and you just put move [? b ?] or something? Would it complain? Would it [UNINTELLIGIBLE]?

CHARLES Yeah, it would complain. Yeah, it would complain. it'll say it's an improperly formed

LEISERSON: instruction, so, yeah. And the other thing you can do, of course, is just try it out. What happens if? That's the great thing about computers. It's easy to do what happened if.

Now, the instruction format is typically an opcode followed by an operand list. So the opcode is a short mnemonic identifying the type of instruction that includes typically the single character suffix indicating the data type. However, for some instructions it turns out you can have two suffixes if the two-- Most instructions operate on data types of the same size. But some of them operate on two different sizes in which case you'll have two suffixes. If the suffix is missing, it can generally be inferred, as I mentioned.

Then the operand list is from zero, two, and very rarely three operands separated by commas. Now, in the architecture manual, in fact, they say if you look at it, they'll show you fourth operand. And I said, four operands? This documentation says there's only three. This one says there's four. I went through the whole architecture manual last night. Every time it says four operands, it says N/A, not applicable. So I think it's just there reserved or something. But anyway there is no fourth operand as far as I can tell.

Now, one of the operands is the destination. And here's where we start to get into some differences. There's actually two standard formats for assembly language that are generally called Intel and AT&T. So AT&T was the original Unix system. And Intel is what Intel uses for their assembler.

They do the destination operand in the opposite order. So AT&T, it puts the destination last. In Intel it puts the destination first. So when you're reading documentation, you can read the Intel documentation. You just have to remember to flip it around if you're actually writing it as we will be using the AT&T format.

Almost everybody uses AT&T as far as I can tell except Intel. So Intel's assembler does it the other way around. And actually now GCC will actually, you can give it a directive to say I'm now switching to writing it in Intel assembler. So you can actually go back and forth between the two if you happen to borrow some assembly

language code from somebody else.

So one of them is the destination. The other operations are read-only, so const in the C++ terminology. They're read-only. So it's always the case that only one of them is going to be modified. And that's the one that's the destination of the operation.

In addition in assembler, there are what are called directives. Besides the instructions, there are directives. So first of all there are things like labels. You can take any instruction and put an identifier and a colon, and that becomes then a way of naming that place in your code.

So, for example, jump instructions want to know to where they're jumping. And rather than having to know upfront what the address is, the assembler will calculate what that address is and everywhere you put x, it'll put in the right value. And you get to name it symbolically rather than as an absolute machine location.

There are storage directives. So, for example, .space 20 says allocate 20 bytes at location x. .long says store the constant 172 at y. It's being stored at y because I said y is here. And asciz gives you a string that's zero terminated. So the standard for strings is zero terminated. You can also, there's one that says give me a nonterminated string. So you can have fun with that if you like that.

The align directive says make sure that as you're going through, so what's happening is the assembler is going through there, is it's laying these things out in memory typically sequentially, the way you wrote it down in the program, in the assembly language program. If you say align eight, it says advance whatever the pointer is of where the next thing is going to be put to be a multiple of eight. And that way you don't run the risk of where you declare a character and then you say, OK, and now I want a long or something, and it's not aligned in a way that the eight bytes correspond to a multiple of eight the way you need to in order for the instructions to properly work on them.

So generally, although, we have byte pointers, most instructions only work on

aligned values. And for some of them that work on unaligned values, they're generally slower than the ones that work on aligned values.

There are also segment directives. So in memory when you run your program, the executing program starts with the program text down at the bottom of memory. And then it has fixed data that's not going to change, static allocation of data. And then it's got heap, which is dynamically allocated data. And then it's got stack. The stack grows downward, and the heap grows upward.

By saying something like text, it says make sure that the next stuff I'm putting goes into the text segment. So that's generally where you put your code. Saying it's in data says make sure it goes in here. So you may want to have a table, for example. So, for example, from pentominos you might have a table there. There's going to be a fixed table. You're never going to change it during the running of the program. Put it in the data segment.

And then there's also things like scope and linkage directives. So saying .global, and you can either spell it incorrectly, as I have here, or with the a, it's the same thing for the GNU assembler anyway. It says make the symbol fib externally visible. And that makes sure that it goes into the symbol table so that debuggers and things can look at it.

And there's a lot more of these in the assembler manual, that link that I showed you to before, tells you what all the directives mean. So that when you're looking at code, which mostly you'll be reading it, making a few changes to it, you can know what things mean.

So the opcode examples, here's some examples. There are things like mov, push, and pop. So, for example, here movslq, this is an interesting one because it's moving. The s says extend the sign because I'm moving from a long from a 32-bit word to a 64-bit word, from 4-bits to 8-bits. So that's why this one takes two suffixes moving from, you'll notice, a 32-bit register to a 64-bit register.

So you have to be careful. This is something I got caught up in the other day. The

results of 32-bit operations are implicitly extended to 64-bit values. So if you store something into EAX, for example, it automatically zeroes out the high order 32-bits of RAX. Because that's the one that it's embedded in. However, that's not true for the eight and 16-bit operations. If you store into an 8-bit field, an 8-bit part of the register, it does not zero out the high order bits of the remainder. So you just have to be careful when you're doing that.

Most of these are things, by the way, that is more cryptic when you're looking at stuff. It's like, oh, how come it's, gee, I thought I had, I'm returning a double word, but it looks here like it's returning a 32-bit word, how come I thought I was returning 64-- Well, the answer is because it knows the high order bits are zero. So it's using the shorter instructions. And yet it still is having the impact on the 64-bit register.

They're all the arithmetic and logical operations. So subtracting, once again, the destination is second. So typically these are two operator things. So you always have the destination occurs both at the beginning and on the left hand side and the right hand side, things like shifts and rotates, control transfer, so call which does a subroutine jump, return from a subroutine, a jump instruction that just says make the next instruction the thing that you're pointing to, and very important, the jump conditionals where the condition is a whole bunch of keys that are things like greater than, less than, and so forth, and different ones for signed and unsigned and so forth.

So typically the condition is computed by using a compare instruction. I probably should have put CNP on here as well, but I didn't. But the CNP instruction is usually what you use to compare two things and then you separately jump on what the condition is.

There's a pretty nice website that has most of these opcodes. However, they only deal with the old x86 without the 64-bit extension. And they use the Intel syntax. But it's really convenient. Because they've done a nice job of making a quick jump table where you can just go, look up the opcode, and pop it up. Otherwise you can just look at them in the manual.

Anyway, that's kind of a convenient place. And, as I say, just beware because it's 32-bit only, and it's Intel syntax. Most of the instructions got extended. I mean it's like, OK, if you do it for eight and 16 and 32, the operation is not going to change that much to go to 64. A few of them do, however.

Now, the operands, Intel supports, the x86, which is Intel and AMD, typically, support all kinds of addressing modes. The rule is that only one operand, however, can address memory. So you have to pick which is the operand that's going to address memory if you have multiple operands. You can't have both operands. So you can't add two things in memory. You always have to take something for memory and bring it into a register and then store it back out.

So the simplest one is two register instructions. Here I've basically marked the-- What have I marked here? I guess I marked-- I don't know. Down here I was marking memory. I'm not sure what I was marking up here. Because they're both registers.

But in any case, this is just adding RBX into RAX. And so it takes the contents of RBX adds it in the contents of RAX. There's something that's called direct. So this is, it says, where you move, x is some constant value, and you move it, the contents of it, into RDI. So if x, for example, is a location that you've stored a value in, you can say move whatever is the value at that location into RDI. Immediate says, which usually is preceded by a dollar sign, says move the address of it, move that as a constant. So x has a value, move that value.

So if you say $3, then you'll move the constant three into RDI. If you said mov3 this, you're going to move the contents of location three in memory. So that's the difference between direct and immediate. So the dollar sign says you're taking that as a literal constant. And the direct says you're actually going to memory and fetching it.

Then things start getting interesting. Register indirect says, in this case, the thing that you're going to access is the thing pointed to by that register. So don't move, in this case, RBX into RAX, move it to the memory location that RAX is pointing to.

Then you can do register index which says, well, it's pointing to it, but I want displaced 172-bytes off of that location, of whatever this is pointing to. So, for example, if you have a pointer to a record, you can then have just a single pointer and address all the fields just by doing register indirect to the different fields using that same register.

Then there is, it actually-- I skipped, actually, a few in here that are subsets of this. This is, I think, the most complicated one that I know. It's base index scale displacement where base and index are registers, the scale is two, four, eight, and if it's not there, it implies one. The displacement is eight, 16, or a 32-bit value. And it says take RD-- oh, I had put the math on here, and then I guess I lost it-- it says take RDX, multiply it by eight, add RDI, and add 172. [WHISTLE].

So, anyway, you can look in the manual. So you'll see some of these instructions being generated. Generally, you're not going to generate these instructions. So when you see them generated, you can see it.

And then, and this is actually new, it's not in the x86. It has this instruction pointer where you can actually access where the current program counter is pointing, where it is in the code, and store that value, in this case indexed by six, into RAX. So you can do it relative where this has to be a 32-bit constant. And what's good about that is it allows you then to write code where you can do things like jump to something that's relative to the program counter.

And that lets you put the code anywhere in memory, and it still has the same behavior. Because you're going relative to where that code is rather than to an absolute location. So it allows the code to be relocatable. So here's-- Yeah, questions, yeah sure.

AUDIENCE: Why was it the index registers, when you have the numbering for the register, what does that mean again?

CHARLES LEISERSON: The number before the register?

AUDIENCE: In the instruction [UNINTELLIGIBLE], that's 60 for RID.

CHARLES LEISERSON: OK, or whatever, whenever it's here, it's basically saying, add that value to the contents of RAX. And so the same thing here, add six to the contents of the instruction. So this is six bytes ahead of me in the instruction stream. OK? So you can actually say, well, what's that instruction ahead of me in the instructions stream? OK?

So here's some examples of essentially the same code and how it gets compiled. So here we're going to have a fou1, fou2, fou3. And in this case we declare x, y, and z to be unsigned integers. We set them to some values. And we just simply say return x plus y or z, bitwise OR with z.

If you look at what the code is that's generated, it says move the constant 45 into EAX. Why does it do that? Well, let's just see. Well, the compiler figures out that it knows what 35, seven, and 45 are. It computes x plus y. That's 41. If you take 41 bitwise OR with 45, it turns out it's masking the same bits, that's 45. So the compiler actually can figure this out that all it has to do is return 45 in a 64-bit register.

Ah, but here it's returning it in a 32-bit register. What happened? It's not obeying the type. The type is supposed to be 64-bits, but that's a 32-bit register. Oh, yeah, that's this thing where it automatically zeroes out the high order bits.

And it uses this instruction, because this is a shorter instruction than if it did the RAX. It could do the same thing with RAX, but it would be more bytes of instruction. So they saved a couple bytes of instruction by doing that. So people follow what happened there?

Let's take a look at the next one. Here it's the same code just let's pass those things as arguments. Well, if you remember the calling convention, parameter one is in RDI, parameter two is in RSI, and parameter three is in RDX. So I don't expect you to remember that off the top your head. But we have the cheat sheet, and you can figure that out.

So here's what it does is, oh my goodness, what is that instruction? This is actually a computation of effective address. So the effective address is basically saying, and it's using one of these funny indexing modes. So what this is actually doing is it's actually adding these two numbers together, the values stored in those locations together, and storing it into RAX. And then it's then OR-ing RDX, what's in RDX, with RAX and then returning.

Remember that RAX is where the result is always going to be. So the result is always returned in RAX. So you can see you have to do a little bit more complicated addressing in order to pull them out as parameters than if it could actually figure out what the numbers are.

Last example here is I declared these things before I ever got their globals. And so I declared them before I ever got in. So that means since they're globals, they have a fixed place in memory. And so the code that's generated is moving, it turns out it allocates them right nearby the instructions here. And so what it does is it has actually a relative offset for x, relativity instruction pointer, put that in RAX, add the offset of x into it, and then OR it with the offset of z, and then return.

And so there the constants are actually stored right nearby in the code so that they can use this relative offset. And the compiler figures out, or the assembler figures out, exactly what the offset is that it actually needs to substitute for y so that it can be a relative offset from the current instruction pointer. Notice that, for example, that's going to change depending upon the value of y here. It's going to change compared to if I accessed y down here. It would be a different instruction pointer at this point.

So it actually just goes but it computes what the difference is so it knows what the distance is. It can compute that at compile time, and then at execution time it just uses whatever constant goes in there. So the important thing here is just to notice that the code depends upon where x, y, and z are allocated.

So the first thing to actually look at good code is to understand the calling convention that's used by the compiler. And here are the basics of it. So the register

RSP points to the function call stack in memory. And the call stack grows downward in memory, like in that little map I showed you before, so that as you push things onto the stack they're getting lower numbered not higher numbered.

The call instruction pushes the current instruction pointer onto the stack, jumps to the call target operand, which is basically the address of the thing you're calling. So when you do a call, it saves your return address on the stack. The return instruction pops the return address off the stack and returns to the caller. It basically says, oh, I know where the return address is. I slam that into the current instruction pointer, and that becomes the next instruction that's executed.

Now, there are some software conventions that are used that's helpful to know. Besides those instruction registers, some of the registers are expected to be saved by the caller, some are expected to be saved by the callee. You're free to violate this in your own little piece of code as long as if you're calling something else, you're obeying it. So you don't have obey this convention in the code you write unless you want to interoperate with other stuff. So if you, for example, have a leaf procedure, you can decide for that leaf procedure, oh, I'm going to make something callee saved that was caller saved or whatever as long as by the time you return you've cleaned everything up for the rest of the world.

So these are conventions. But for the most part, you're not going to violate these, and the code that the compiler generates doesn't violate these because it expects everything to interoperate.

So here's how the subroutine linkage works. We're going to do an example here where function A calls function B which will call function C. And right now, we're at the point we're executing B. And so on the stack are the arguments that were passed from A to B that did not fit within the registers. So normally most of the arguments are within registers. But if you exceed the six registers then, because you have a long argument list, then it gets passed on the stack. And here's where it gets passed.

The next thing is B's return address. This is the thing that got smashed in there

when you did the call. It got pushed onto the stack. And then there's what's called a base pointer for A. And this is the way that A ends up accessing its local variables. And then there's a separate region here where it's going to put arguments from B to B's callees if they exceed the six registers, if any of the things that B is calling require more than the six registers.

So let's just take a look. So function B can access its nonregister values by indexing off of RBP. So these we say, these are in a linkage block. And the reason is because it's actually part of A's frame as well. It's a shared part of the frame where A stores it into memory and then B's going to fetch it out of memory. And that's the linkage block.

So this is positive in memory. So if I use a positive offset, I then go up to getting the arguments from A. Then it can access its local variables from the base point with a negative offset because we're growing down in memory.

Now, if it wants to call C, what it does is it places the nonregister arguments into the reserved linkage block here, which are arguments from B to B's callees. And that once again acts just as if they're local variables. It's positive index off of RBP, sorry, negative offset off of RBP. So it pushes those things into the argument, into that region, if it needs to use that region.

Then we actually, once it's done that, we have the call. So B calls C which saves the return address for B on the stack, so it saves it on the stack, and then transfers control to C. So now it starts executing C's code. And what does C do?

So C is going to have to advance these pointers to refer to its region rather than B's. It does it by saving B's base pointer on the stack. So it saves this pointer here so that it can restore it when it returns. It advances, it sets its new base pointer to be where the stack pointer is now and then advances the stack pointer to allocate space for C's local variables and linkage blocks. Watch, here we go. So that ends up being C's frame. So notice that B's frame and C's frame are overlapping in the linkage block between them.

Now, if a function never performs stack allocations except during function calls, there's a great compile time optimization that the compiler will often do. And what it will do is realize that this distance is constant. So, therefore, it doesn't need RBP. It can just do the math and index everything off of RSP as long as RSP is always the same, for example for C, when C is executing.

There's certain C commands like [? alaka ?] which changed the stack pointer. If you use those, the compiler can't do that optimization. But if the storage on the stack never changes for a given frame, then it's free to make this optimization. So you'll see code where RBP has been optimized away. How about some questions before we go on and do an example. Yeah?

AUDIENCE: [INAUDIBLE] should there be A's return address, where you try to [INAUDIBLE]?

CHARLES LEISERSON: Up? Oh, this should be, sorry, this should be A's return address. Yes, you're right. OK, good, typo. Is somebody catching my typos to-- OK, yep, a good one, that should be A's return address. Sorry about that. This is B's return address. Any other questions? That's good. That means you understand something. Hooray.

So let's do an example. So here's my fib example. And I compiled this with minus oh zero. Because when I compiled it with minus oh three, I couldn't understand what was going on. So I compiled this with minus oh zero, which gives me really unoptimized code. And that lets me be the compiler optimizer.

So here's the code that it generates. So we can take a look at a few things here. First of all is declaring fib to be a global. And it's got some other things here. I actually took out some of the directives that were in here that were irrelevant for our purposes. If you actually compile it, there's a lot more directives that are stuck in there and a lot more labels and things that you don't need to understand it.

There are two labels here. And so you can see here basically what's going on is we're first of all doing the advancing of the base pointer and advancing the stack pointer here. That's doing that operation that I showed you, those of moving the base and stack pointer up. And then at the end here this is equivalent to doing, a

leave instruction is equivalent to undoing that. So Intel lets you do one leave instruction rather than making you put these instructions in every time. It's exactly the same thing.

But in any case, let's just sort of see what's going on here. So we're pushing some storage. This is saving a register here. We're then advancing the stack pointer to store 24 bytes of temporary storage. And then we're start to do some computations here. This looks like we're comparing one with something and then doing a ja. So this is a jump above. This is the unsigned version.

What you're looking is to see, here we say if n is less than two, in fact, what it's doing is saying if n is greater than one go to L4. So it's actually doing the other one. So you can see then L4 is, what happens is the one that has the two calls to fib, recursive calls, so that's this part of the code, and it's doing that if it's greater than one. And otherwise it's going to execute these instructions, which are basically returning n. And so it basically does some computations. And then both of them converge here where it moves the results and then pops it off and so forth.

So that's sort of the outline of what's going on there. So let's dive in here a little bit and sort of see what's going on, see if we can read this a little bit more closely and whether we can optimize it. So the first thing that I noticed in looking at this is look at all this memory addressing that we're doing. What do you suppose this thing is, minus 16% RBP? So this is the base pointer. So this is a local variable because it's a negative offset off of the base pointer.

What do you think it's doing there? What's stored in here? Yeah, this is where n is being stored. Because what are we doing? We're trying to compare n with one here even though it says two up there. We're comparing it with one here.

And so I look at that, and I say, look, I'm comparing it with one, then I'm jumping to L4, then I jump to L4 or not. And then let's say I don't. Well, then the first thing I do is I move n into RAX. But wait a minute, I just compared it with that. So I'm accessing n again. I'm accessing it a third time.

How about if I try to store that stuff in a register instead? So what I did is I picked the RDI register, because that one happens to be available, and I said do they, if you look here, what did we do? We stored RDI, which is the first argument, into memory. And then we compared with it in memory. Why don't we compare with it in RDI? Right? Duh, stupid compiler, well, because I had minus oh zero.

OK, so I can do that improvement. So what I did was I edited it to put RDI there and RDI here and RDI here. And I went up and I said, what about RDI here? Why didn't I replace that one? No, there's no loop going on here. It's recursion.

AUDIENCE: [INAUDIBLE PHRASE]

CHARLES LEISERSON: Yeah, the problem is that when I call fib, RDI gets garbaged on me. Because RDI is going to be the first argument to-- See it's being garbaged here? It's garbage as far as my use of it for n. It's being used to pass n minus 1 as the argument to the recursive call. So I can't replace this one after fib because RDI no longer has it. Because I had to leave it.

But even so I went from 5.45 seconds for the original code to 4.09 seconds when I compile that just with that little change. I felt pretty good. I felt pretty good. So then I wanted more. That was fun. I wanted more.

So what was the next thing I noticed? I noticed that-- And by the way almost all the things, that stuff I did last night. This is what I did an hour before class so we'll see whether it-- So then I noticed that, look, we're moving this stuff here. We keep using minus 24. And once again, memory operations are expensive compared to register operations. Let me try to get rid them. What do you suppose is in here?

So look, we move RAX into the local variable minus 24. And then we jump to L5. And we move minus 24 into RAX. That seems kind of unnecessary. Here we move RBX into minus 24. Then we move minus 24 into RAX. What is this value first of all that I'm storing there? What's going to be in RAX at the very end?

RAX is the return value. So I'm trying to save, here in this case, this is the branch where I just want to return n. I just want to put RAX to have it return, be in RAX

when I return. So I've got the value here. It's just n. It was in RDI. It's now in RAX. But that's clearly unnecessary. Why go put it into memory and then take it back out again? And here just put it in RAX directly.

So that's what I did. I basically, instead of moving it here, I changed this instruction that said add it and put in RBX, I said, no, don't put it in RBX. Let's just add RBX into RAX, and then it's right there. And this one, get rid of those so that it's now moved into RAX and it's in RAX. So I did that. I dropped to 3.9 seconds. That felt pretty good, too.

In addition, I got rid of this extra variable. So now I could actually reduce my storage requirements. However, when I measured it with this being 24 and not being 24, it was the same speed. So it's like, eh, but I didn't want to waste the storage anyway.

So then I looked a little bit further. And I noticed that I want to get rid of this access to n here. So basically I'm subtracting it, and I'm storing n. How can I get rid of it? And this took me a little while to figure out.

What I realized is, look, we're storing stuff away in RBX. We have RBX as an available register because I saved the value of RBX with this push instruction there. So RBX is an available register. We're using it to keep the return value of the first call to fib. So I'm going to use it for their first call to fib so that when I make the second call to fib I can then add the two things together.

Well, how about if before the first call the fib, why don't I use it to store the value of n and then use it to store the value of the return value of fib of n minus 1? So I did that. And that took a little bit of moving things around a little bit. But I managed to get rid of it by using RBX for two different purposes, one to store the temporary, the value of n, and the other to store the return value when I need it.

And when I did that, I got it all the way down to 3.61 seconds. I actually ran it with minus oh three, took about two seconds. So I think I can keep my day job. But kind of fun to go in and sort of see what are the things that can be done. And you can get a very good sense of what's going on.

The more important thing is when you look at compilers generating your code, as we saw on the last lecture on profiling, you can see, oh, it did something silly here. So you can actually go and say, oh, it's doing something silly. We can do a better job than that. Or, oh, I didn't realize I'd declared this an int when in fact, if I declared it a unsigned int 64, it actually would produce faster, better code. Yeah? Question?

AUDIENCE: Sorry. So when you said that when you run it with minus oh three, what you're just saying is even though you optimized the [INAUDIBLE]?

CHARLES LEISERSON: As the compiler, that's why I said I can keep my day job. So simple optimization strategies, if you're playing with things, is you try to keep values and registers to eliminate excess memory traffic. You can optimize naive function call linkage. And the most important thing probably is constant fold. Look to see where you've got constants that can be combined together. There are other optimizations that compilers do like common subexpression elimination and so forth. But these are sort of the ones, if you're doing it by hand, these are sort of things to focus on, particularly number one, just get rid of excess memory traffic.

Let me say, by the way, in doing this I also went down a bunch of dead ends, things that I said, oh, this should definitely save, and then it was slower. And then we look at it, and it turns out, oh, my branch misprediction rate is going way up and so forth. That's why you have a profiler. Because you don't want to do this blind.

Now, how does the compiler compile some common high level structures? So if you have a conditional, for example, if p, do the ctrue clause, else do the cfalse clause, what it does basically is it generates instructions to evaluate p. And then it does a jump with the condition to see if p is false to the else clause and executes those instruction. And then otherwise it passes through, does the true clause, and then jumps to the end. And you'll see that pattern in the code when you look at it. So that's a very common pattern for doing conditionals.

Compiling while loops is kind of interesting because most while loops start out with a jump. So here are the instructions for the body of the while loop. And here's the test. And what they usually do is they jump to the test, they evaluate the condition, and if

it's true, they jump to the loop. Otherwise, they fall through. And then they go back, do the loop sometimes for the first time, et cetera. So that's kind of the pattern for a while loop.

For a for loop, they basically just convert it into a while loop. You basically take the initialization code. You execute that. Then while the condition is true, you do the code followed by whatever the next code is. And so it ends up converting for loops into while loops.

Now, arrays are, how do we go about implementing data types? Arrays are just blocks of memory. So you can have basically three different types of array depending upon where it gets allocated, either allocated in the data segment, allocate on the heap, or allocated on the stack. Sometimes even the static arrays these days can be allocated in the code segment, if you're not going to change them.

So one thing is to understand that arrays and pointers are almost the same thing. If you have an array, that's a pointer to a place in memory where the array begins. And a zero is the same as the value you get when you dereference the pointer to a. A pointer, if you think about it, is actually just an index into the array of all memory. And the hardware allows you to index into the array of all memory. Well, it can also allow you to index into any subregion of that memory. And that's why arrays and pointers are basically the same thing. Here's a little quiz. What is eight of a?

AUDIENCE: The a elements?

CHARLES LEISERSON: Yeah, it's basically a of eight. It's basically a of eight because the addressing that's going on is essentially the same. Even though we prefer to write it-- If you start writing code like this, I guarantee that at some companies they'll get very angry at you even though you say, it's the same thing. But what's going on is you're actually taking the base, the address of a, you're adding eight to it, and then dereferencing that value.

And indeed, even in C, they actually do all the coercions, even if eight is a different

type, it actually does the coercions properly so that they are actually the same thing. Because it does them after it's converted it into a dereference of a plus 8. So it's kind of interesting that it works right even though when I looked at that I say, well, what if it's bytes verses words and so forth? Yeah? Question?

AUDIENCE: Will there be a performance difference between putting data on those three different arrays?

CHARLES LEISERSON: Yes, there can be. In particular, static array, it knows exactly where the base pointer is as a constant. Whereas the others it has to actually figure out where it is in the heap you need a pointer to it to dereference it. It can't put it right into the instruction stream itself.

AUDIENCE: [INAUDIBLE] those would be just constant [INAUDIBLE], right? So I can either put that in the static array or--

CHARLES LEISERSON: Yes, generally it's faster to have it in a static array if you can. I want to finish up here so that-- We have structs. Structs are just blocks of memory also. So you can have a bunch of things here. This is a bad way to declare a struct because the fields are stored next to each other generally in the order you give them. So here it says it's x and then i and double.

What happens here is you have to be careful about alignment issues. So if you do [? char, ?] it's then got to pad it out to get to the next alignment for an int. And it's got to pad that out to get the next alignment for a double. Whereas if you do it in the opposite order, it starts packing them. So generally it's best to declare longer fields before shorter fields because then you know when you're finished with the longer fields, you're already aligned for shorter fields.

Like arrays, there are static, dynamic, and local structs. So that's all they are. And so you'll see in the indexing--

There's also stuff that-- and actually this is important for one of the binary puzzles we gave to figure out what it does-- there are what are called SIMD instruction. This is single instruction multiple data instructions where a single instruction operates on

multiple pieces of data. They operate on smaller vectors. And there are 16 128-bit XMM registers, which you can view as two 64-bit values or four 32-bit values. And you can do an operation on it. So this is used for multimedia, for streaming applications, and so forth where you're trying to shove a lot of data through and you're doing the same repeated stuff on the things at time.

So there are instructions that operate on multiple values. For example, here we're moving four 32-bit ints into this particular XMM register. And similarly here's another one, we're adding it. And you can look at the manual for these. So you may come across these because they're using up parts of the machine. Mostly those are the kinds of things we can say look it up in the manual, because nobody's going to remember all those instructions. Of course, if you get a job with one of the graphics companies, then you may become very familiar with these kinds of instructions.

There's a lot more C and C++ constructs that we don't have to go into. You can have arrays of structs versus structs of arrays. And there can be a difference in performance. If you have an array of structs, then it makes it, if you're accessing one struct, you can access the other structs very easily.

But if you're using things like the SSC instructions, then it may be better to have structs of arrays. Because then when you're access an array, you can stream what's called a stride of one, a regular stride of just one memory location after the next, to do the processing. So the hardware doesn't work as well if you skip by seventeens to gather things compared if you just get one thing after the next. Because there's prefetching logic that tries to fetch things from memory faster.

There are things like function pointers. So you can have store function pointer into something and then call that function indirectly. There's things like bit fields in arrays. There are objects, virtual function tables. We'll get into some of these when we do C++. And there's a variety of stuff having to do with memory management that we'll talk about.

But this is mainly to get you folks sort of at the level where you can sort of understand and feel comfortable with dealing with the assembler. And you'll see that

those resources are pretty good resources. But the basics are relatively simple, but it's hard to do it without a manual or some online reference material. Any questions? What are the first two lessons I taught you today? Number one is--

AUDIENCE: [INAUDIBLE]

CHARLES LEISERSON: --write tests before you write code. And what's the second lesson?

AUDIENCE: Pair programming.

CHARLES LEISERSON: Pair programming, not divide and conquer, I teach algorithms where divide and conquer is a fabulous technique. With programming, pair programming is going to have you generally get where you want to get faster than if you're programming alone. OK, thank you.