PROFESSOR: So today we're going to talk about bit hacking, which is a topic that has a long, long history in computer science. We'll only cover on a few of the techniques. So let's just get going.

So I want to swap two integers. So I think most of you would know how to write a program to swap two integers. And it would look something like this. And mostly this is pseudocode. I'm not going to be doing declarations of types and writing full code, in order to make sure things get on slides and so forth.

So what do you do? You assign a temporary to the value of x. You then let x take the value of y. And then you let y take the value of the temporary.

What could be simpler? Well how about doing it without a temporary? So how do you swap two numbers without a temporary?

So here's one way. So what's going on there? So the carrot is an XOR, exclusive or, OK? So here's what's going on.

So let's do an example first. So I have x and y. I then let x be the XOR of x and y. So, as you see, that first bit is the XOR of one and zero. The second bit is the XOR of zero and zero, which is zero. The third bit is the XOR of one and one. That's zero. And so forth throughout the bits.

So then I let y be the XOR of x and y. And then, finally, I let x be the XOR of x and y again. And now, if you notice, that number is the same as that number. And that number is the same as that number.

Magic. We're going to see a lot of magic today actually. OK? We're going to see a lot of magic today, no temporary.

Why does this work? So the reason this works as a great property of XOR is that it's its own inverse. So if you take x exclusive or y, and you exclusive or that with y, you get x. If you were to exclusive or that with x you would get y. So that first step is basically putting in here the XOR of x and y so that when you end up on the next step computing the XOR of y and this, you get x.

So now you've got x here. And you've now got the original x XORed with y here. So to get back the value of y you just XOR out the x. So it swaps them. Whose brain hurts?

You can study these later on. But a pretty neat trick, pretty neat trick. Does it perform well? Turns out not really. And the other way is actually a better way of doing it, generally, with most compilers and architectures. And the reason is because the other way of doing it you can actually, essentially, pull two things out of memory, one into a temporary one-- and then stick them back like this. That's what the compiler ends up doing.

Where as this one, it has to wait for each step. And so you don't get to exploit instruction-level parallelism. Remember from last time instruction-level parallelism is the fact that a processor can issue more than one instruction at a given step. And here, this sequence of operations, each step has to wait until the previous one is computed before it can execute. So you get no instruction parallelism in this.

So it's not particularly high performing. But there are other places where we'll use this kind of property. But it's a neat bit hack, swap two things without using a temporary.

Now here's a real bit hack. And a real useful one. Finding the minimum of two integers, x and y. Gee whiz, let me just call a sub routine or something. So you might be tempted to write something like this; if x is less than y then the result is x. Otherwise, the result is y. Seems pretty straightforward. Or if you know a little bit more c, you can write it with this sort of cryptic if x is less than y, then x else y. So those are two equivalent c ways of doing things.

So what's wrong with that? Well nothing if you don't mind slow code. In fact, for something like this the compiler actually will optimize it to deal with it. But let me just point out a couple things.

First of all, the processor has within it a branch prediction unit. Whenever it comes to a branch it guesses which way the branch is going to go, and proceeds to speculatively execute along that path. If it turns out to be wrong it says, whoa, hold your horses, got to go that way.

To do that, it empties the processor pipeline. And that takes, on the machines we're using, around 16 cycles. So you don't want to have branches that are mis-predicted. And, in particular, what you want to look for in conditional instructions, is whether they're predictable.

So something that's almost all the time branching the same way, that's a predictable branch. The hardware is very smart about figuring out how to predict that, and will make the right prediction. And you won't pay any performance penalty. But if you have something where you don't know which way it goes, so in a code like this, the architecture isn't going to know. If you just throw at it various pairs of x and y it's a 50/50 guess as to whether it guesses right. So half the time it's going to predict the wrong thing.

So the compiler might be smart enough. But maybe not. But you can be sure. And here's a way of being sure. You write this code. What is going on there?

So here we go. We're taking x less than y, and taking a minus sign. Yikes, what's that do? Well c represents the Boolean's true and false with the integers one and zero, respectively. So if you execute the operator x less than y, as opposed to doing a conditional based on x bit less than y, it returns to either a zero or a one, depending upon whether it was successful.

So when you take the negation of that, this is either zero or one. It's either going to be zero or minus 1. And what is minus 1 in two's complement arithmetic? It's a word filled all with ones. So you either get a word all filled with zeros or you get a word all filled with ones. So if x is less than y you get a word all filled with one's.

So then what are you doing? You're doing x XOR y, and you're ending it with all ones. Well that's a noop to end it with all ones. To mask something-- if I do an and of anything with one, I get whatever the thing is. So this expression ends up evaluating to just x XOR y.

Great, then what? I take y here and I XOR it, I get back x because of that inverse property of XOR. So if x is less than y, then r gets the value of x.

If x is greater or equal to y then this expression evaluates to zero. And a word of zeroes ended with x or y gives you a word of zeroes because zero is an annihilator for and. Wherever you and with zero, it doesn't matter what it is. You get zero. So therefore this just becomes r equals y because y XOR zero is y.

So pretty clever, is this really better? Seems like an awful lot of operations. Well the answer is yes it is, because all of this goes on within the processing unit rather than with anything having to do with memory. It gets the values for x and y to begin with and then it's all instructions within the processing unit. Those typically take one cycle. And if there's any parallelism in it the parallelism will be able to execute even more than one operation per cycle.

In fact, the machines we're using have are six issue. They can run six operation simultaneously, each taking a cycle. So the difference between that and going out to memory is really quite considerable. So everybody follow that? Pretty cute trick, how to make it go fast. Yes, question?

AUDIENCE: Doesn't the expression that's tested, doesn't it have to be [? weighed for ?] the inner expression before you take--

PROFESSOR: There's no-- this is a comparison. This is operated like a-- so there's no compare instruction there. It's a CPU operation. It's an arithmetic and logical operation of the CPU that it can do in one cycle, is to compare. The normal thing that you're trying to do if you have an if is you're trying to change the program counter. And that's what's costly. Not the actual doing the test of the branch, test of whether x is less y. OK?

AUDIENCE: Would you have to wait for that to finish before you can do the negation?

PROFESSOR: Yes you do. So that's one cycle, two cycles, we can add it up here, three cycles. This can be going on in parallel. So it's really only two cycles total. Three cycles, four cycles, so in four cycles you can get the minimum done.

The L1 cache in the architecture we're using costs you four cycles to fetch something if you get a cache hit in the L1 cache. That's the cheapest memory operation you can do, is four cycles. This computed the whole minimum in four cycles.

Here's another one, modular addition. So sometimes you know something that the compiler doesn't know. Like suppose that you know that x is between zero and some value n, and y is between zero and some value n, and you want to compute their sum.

So what is that the sum is going to be less than what? 2n. So normally a modular operation is very expensive because it involves a divide. Now multiply is normally more expensive than an ordinary ALU operation that's just a bitwise operation, like addition, or XORing bitwise XORs, or comparison, or what have you. Those are very cheap one cycle operations.

Multiply is usually a many cycle operation. Divide is often implemented by doing repeated multiplies using any of a variety of techniques, including Newton techniques. Sometimes there is a divider, or a divide step. But divide is, generally, in any case more expensive, even though it's doing operations all within the processor.

So if you actually compute mod using your percent thing. This actually can be quite expensive unless you're dividing by a power of two. If you mod a power of two that's easy because the processor, if knows it's a power of two, if the compiler knows it's a power of two, it'll just do a masking operation on the low order bits to give you whatever the remainder is, mod2 to the n.

But if you're not in that situation, n may not be a power of two, you still want to do something modn, there still are some tricks you can play but the compiler won't- this is one the compiler generally won't play for you because the compiler won't know that these are preconditions in your code.

By the way, one of the most common things is just doing x plus 1 modn. Very, very common thing to be doing, x plus 1 modn, where you're wrapping around in some index space.

Here's another way you could do it. So divide is expensive. Here you could just say z equals x plus y. And then if z is less than n, give z otherwise z minus n. The problem with this is that it's got an unpredictable branch. To execute this code, I could have written it out with an if statement, it's got to change the program counter to execute either this or this. so not very fast because you have an unpredictable branch. And we already talked about that has to empty the pipeline if it's wrong in the guess.

So here's a way of doing it which doesn't have an explicit branch. So we compute x plus y. And now what we do is we look at whether z is greater than or equal to n. And if it is, we're basically going to take the negation. So that if it's is greater or equal to n, the negation here is all ones, once again, is minus 1. And so this becomes n and-ed with minus one. That gives me n. And so then I'll take z minus n. That's what I want

However, if this is z is less than n, then this will evaluate to zero. Minus zero is zero. And n and zero is zero. And so I'll end up just getting the x plus y. So it's basically the same trick with a couple of twiddles on the minimum that we saw on the previous foil. Who's having fun? Good. As I said, we're going to see lots of tricks, magic tricks even.

Round up to a power of two. This is a common thing that you want to do here. This, for example, goes on in memory allocators, which we'll talk about later in the course. So in a memory allocator, somebody asks for a hunk of storage of size 19, most memory allocators want to give out chunks that are powers of two for reasons we will discover later.

So you want to round up to the next higher power of two. So how do you do that? So here's an example. So what I do is I decrement n, and then I update n or-ing it with the left shift of n, and then or-ing that with- sorry, the right shift of n by one. Then the right shift of n by two, et cetera, et cetera.

So here's an example. So here's my original number. And what I want is to round it up to the next power of two. This is what I'm going to end up with in the end. See I've got the next higher power of two? Just one bit is on if I've rounded up to the next higher power of two.

So what do I do? I basically decrement and then I take this word and I shift it by one to the right and or it in. And then I shift it by two and or it in. And then I shift it by four and or it in. And then, in fact, I shift it by eight and or it in. And I didn't do that. And since this isn't a 64-bit word I skipped the last two instructions.

So what's going on when I'm shifting and or-ing it in, by one, by two, by four. What's happening? Yeah?

AUDIENCE: [INAUDIBLE] this way there's no number of ones starting from the one.

PROFESSOR: Yeah basically, from the most significant bit, if you look at what's happening with the most significant bit, you're shifting it by one. Then you're shifting it by two. You're flooding the low order bits with ones. And because it's an or, as soon as something gets set to one, it stays a one.

So it doesn't matter what's actually happening in the low order bits. The only bit that we care about is this bit. And it basically floods all of the other bits with one. And then, once I've flooded them all with one I increment. And that gives me a carry out to this position and gives me the next higher power.

So why did I decrement here, and then increment there? What's the decrement for? Yeah?

AUDIENCE: So that you can flood with one because you want to get yourself back. So you want to the add the one--

PROFESSOR: But why did I decrement first?

AUDIENCE: If you were not to decrement, it would just flood everything with ones.

PROFESSOR: Well here, if I didn't decrement right here, I would have gotten the same result. If you already have a power of two. If I already have a power of two and I flood the low order bits, then I increment, I'll get the next higher power of two. So by subtracting one I make sure that I'm handling that base case when n is a power of two. Yeah?

AUDIENCE: Does the [INAUDIBLE] operate [INAUDIBLE]

PROFESSOR: It actually, the compiler is not going to care in this case. But it does make sure that it doesn't bother to try to return and keep around the old value. But it's smart enough to not worry about that. Yeah, I mean, some people like post-fix decrementing, and some people like pre-fix decrementing and it doesn't matter in most cases. But sometimes doing it after-- there are situations where doing it post-decrementing costs you a cycle. So everybody got the idea here? So basically round up to a next power two.

How about computing a mask of the least significant one in a word? So I want to mask which is the power of two, the word that's all zeros except for one in the least significant one bit. Any ideas how to do that?

This is a classic trick. Everybody should know this trick. You take x and you and it with its two's complement. Take x and and it with it's two's complement.

Why does that work? So here's x, some value here. The two's complement is the one's complement plus one. Right? If you remember. So one's complement just means I compliment every bit. The two's complement is I compliment every bit and add one. So when I compliment every bit and add one, basically I go all the way to the least significant bit and then I get zeros after it.

So right up to there I get-- it's one's complement and then it's basically- would have been 0, 1, 1, 1, 1, 1, plus 1. The carrot pulls you back up to there. And so then when you and them together, oh look at that. There's our least significant bit sitting there. Pretty good one?

So so how do you find an index of the bit? So by an index I mean this is bit 01234. Well it turns out these days many machines have a special instruction to do that. And so if you look around and you find the right library that calls that instruction, you can use that instruction pretty cheaply.

But there's still a lot of machines, especially things like Mobile machines, et cetera, where they have a depleted instruction set. Where they have no instruction to convert from a power of two to essentially it's log base two. So LG is the notation for log base two.

So how do you go about doing that? So what we're going to do is do some magic to motivate the solution to this. So one way to do it is to use the ESP instruction. Are people familiar with the ESP instruction? What's that? The stack [? order? ?] No, no, that's BSP on some things. Or yeah, right, so in the extended instruction set-- yeah, OK. Yeah, so the ESP instruction; Extra Sensory Perception.

And we have today the tremendous magician Tautology who is going to demonstrate the theory behind finding the index of the bit. So please give a warm hand for Tautology.

TAUTOLOGY: How about now? Can everyone hear me? So, as my good friend Professor Leiserson has mentioned, I am the amazing Tautology. And today I am going to show you an amazing card trick which will baffle your minds for approximately five minutes until he shows you the next slide. All right, but to do this I will need five volunteers from the audience.

PROFESSOR: Who can follow instructions. [LAUGHTER] So who would like to volunteer to participate in real magic? Here we go, one, two, three, four, only four? We need one more. So come on up and line up along the front here. We need one more, one more volunteer. One more volunteer, you get extra points. Remember participation is part of your grade. OK I'm going to cold call. Here we go.

TAUTOLOGY: All right, excellent. Please cut this deck.

PROFESSOR: First you gotta show it's a random deck.

TAUTOLOGY: All right, I will first show you that's it's a random deck.

PROFESSOR: It's only 32 cards. It's only 32 cards. So he pulled out some of the other cards. But they're in a pretty random order there. So we want to give everybody a chance to shuffle the deck here by doing a cut. Look at it, but don't show it to Tautology. Why don't you go around the back so that the class can see the cards? So hide your cards while he runs around behind. And then turn them around so that the class can see what the cards are.

TAUTOLOGY: All right, are you guys ready to turn around? Cool, excellent.

PROFESSOR: I'm not going to look at them either. There are no dupes. This is all done by ESP.

TAUTOLOGY: All right, now-

PROFESSOR: Why don't you come over here where you can see the class?

TAUTOLOGY: Hello everybody. I'm going to tell you what is on those five cards, which I have not seen. Behold I have not seen them. OK, are you guys ready?

PROFESSOR: Now you've got to think about it. You've got to think hard about what your card is. If you're not sure you can check. But you've got to think real hard about what your card is.

TAUTOLOGY: Okay.

PROFESSOR: Are you guys thinking hard? I think you need some technological assistance.

TAUTOLOGY: I could use some technological assistance.

PROFESSOR: There we go. This is our brain amplifier. It amplifies the brain waves coming from you. OK, give it a go.

TAUTOLOGY: Hang on, is this thing on? There we go. Now it's on. OK, here I go. All right guys I'm having kind of an off day. So I'm going to need a little bit of help. Can you guys just raise your hand if you're holding a red card? So that's a red?

PROFESSOR: Red, red, black, red. They did it right, right?

TAUTOLOGY: Now let me give this a shot. So red, black, red, red, red.

PROFESSOR: No, red, red - yes OK you do it your way.

TAUTOLOGY: Now, now I might be wrong. But I think- am I seeing a diamond with a seven on it? Is that what I'm seeing?

PROFESSOR: Oh! How impressive is that?

TAUTOLOGY: OK, one down, four to go. What else am I seeing? I believe I am seeing. You're going to have to think about this card pretty hard. I'm having an off day. So I think I'm seeing a spade, with a six. Thank you for your honesty. Thank you. It does mean a lot to me. All right, so I'm seeing a- now what am I seeing now? What could this be? It's some kind of a heart. I think, just maybe, it has a value of five.

PROFESSOR: Oh! Three in a row! I don't think he's doing this at random. There must be ESP at work here.

TAUTOLOGY: Clearly. It's all the hat to be honest. It's all the technology, just the latest technology. All right, so-

PROFESSOR: It gets harder as we go.

TAUTOLOGY: It does get harder as we go. But, what is this? do I- I think- no it can't be- can it be? The three of hearts? It's the three of hearts! My old nemesis the three of hearts. OK just one more card. [LAUGHTER] Did you just swap cards? I think I just watched you swap you cards. Well I'm just going for in the set-

PROFESSOR: And they did it without a temporary notice. They must have xor-ed them together or something.

TAUTOLOGY: So the last card, which may or may not be in the last person's hands. Let me see if I can see it. What could it be? What could it be? No, no! No, anything, not the- not that! Not the 6 of diamonds!

PROFESSOR: All right, five out of five! Thank you, thank you. OK guys go back to your seats. So that was a pretty easy trick, right?

TAUTOLOGY: Yeah, pretty easy.

PROFESSOR: So how does it work? What's the basic idea behind it? What's the basic idea behind it? So key thing is how many bits of information did he get? Five bits of information. And there were how many cards in the deck? Thirty-two. So the pattern of the five bits, of which cards were read let him know where in the cyclic sequence of cards he was, right? Because the cards really weren't random. They just looked random.

And so five bits is enough. But that means that every the sequence of five cards in that deck, as you rotate it around, has to have a different bit pattern. So does anybody know a name for that property? A circular sequence that has that property?

The property is that if you- well, let's see. Maybe I have it up here. So here's our magic code, which is going to compute the log of x. And it's using what's called a De Bruijn sequence. So let's come back to the magic trick in a minute. And let's look to see how we compute this. And then we'll understand how both work.

There's a magic number here called De Bruijn. De Bruijn was a Dutch mathematician. And then there's this funny conversion table. And to find the log of x, where x is a power of two, I multiply x by this magic number here. Right shifted 58 places. This is keeping how many bits after the multiply here? Six bits because it's a 64-bit word. And then looking it up in this table.

So let's take a look at what's going on there. So a De Bruijn sequence, s, of length 2 to the k, is a cyclic zero one sequence, such that each of the two to the zero one strings of length k occurs exactly once as a substring of s. That's a mouthful.

Let's do an example, smaller. So for example k equals 3. So here's a sequence, 0, 0, 0, 1, 1, 1, 0, 1, base 2. If I look at the first three bits it's 0, 0, 0. The second three bits is 0, 0, 1. And notice that as I go through here every sequence, and these are wrapping around the end, taking the last bit and then the first two of the end. Every one of these gives you an index of every different bit pattern of length 3.

So these came up because people played with those keypads where you have to enter a combination, right? And if you have a keypad and you want to enter a combination- let's say the keypad has only two numbers on it, 0 and 1. And you have to hit the right sequence of k numbers. So the naive way of doing it would be to say well let's try 0,0,0,0,0,0,0. Then let's try 0,0,0,0,0,0,1, then 0,0,0,0,0,1,0.

So what that'll do is you'll have to go through 2 to the k numbers, each of which is k bits long, for k times 2 to the k punches in order to be sure that you've hit every number to open the lock. The De Bruijn sequence takes it from k times 2 to the k down to 2 to the k. Still exponential in k, but there's k at the front because it's making it so that each sequence that you have nests into the previous one. And that's basically what's going on here. Every sequence of length 3 exists in the De Bruijn sequence. And so this one here is a De Bruijn sequence of length 64 as it turns out.

And what we had in the magic trick was a De Bruijn sequence of length 32. So that when you cut the cards and you looked at the first five cards that was a unique pattern of reds and blacks. It told you where you were in the rotation of the sequence. And then there's a little bit of cleverness to how it is that you translate that into cards. Because remembering 32 cards, and what their sequence was, and so forth, that's pretty hard.

But it turns out you can just do an encoding of the five bits. Two of the bits encode the suit. So the high order bit encodes the suit, the two bits encode the suit, and then the last three bits tell what the number is, 1 through 8. So that's how that worked. It wasn't really magic after all. Who's surprised?

So how can we use this in this particular code? So for this, basically the convert table does the following. It says, well, if you've got zero, the offset, the shift of this amount here, is zero. And if you've got one, then the shift is one. And if you have a six- where's six in here? Sorry if I have a two here, then that's going to be that I'm six.

So this table is inverting this number. Do people see the relationship there? So if I know what the pattern is I can do a look up and tell how much did I shift by? If I am shifting by a given amount there, or circularly shifting.

So here's the way that code works. Let's say we've got a number like 2 to the fourth that I'm trying to figure out what the exponent is. It's always a power of 2. So I'm looking at 2 to the fourth and I want to extract 4. But all I have is the mask that's 16, which has the one bit on. What I do is I multiply this number, the De Bruijn sequence number, by 16.

Well what happens when you multiply by a power of 2? It shifts it by 4 bits. So it's shifted the bits by 4 bits. And now if I right shift it by 8 minus 3, I capture the top 3 bits. In this case, 1, 1, 0, which is 6. Then I convert 6. It says I had a shift of 4. And just with 64 bits it's a longer De Bruijn sequence.

So it's performance is limited by the fact that you have to do a multiply and a table look up. But it's generally fairly competitive for many machines that do not actually have a log base 2 of a power of 2. These days machine instructions are getting- there are instructions that will do that in a single instruction for you. But if you don't happen to have one on your architecture and need to do this fast this is a reasonably fast way to do it. Even with a table look up and the thing.

The other way of doing it, of course, would be to shift by one, shift by one, shift by one, until you get the one. And there's some other techniques as well that you can use. You can do divide and conquer in a binary way, where you do binary search for where the bit is, by shifting and so forth, and hone in.

But the problem with those techniques, the binary search in particular, is what? If I try to binary search to find a bit what's that going to be? Yeah, branching. You're going to have unpredictable branches. And each of those will cost you 16 cycles. And so with a 64-bit word you've got 16 cycles times six bits that you're trying to decode, times however many instructions it actually takes you. It adds up to a lot of cycles. But that can sometimes be an effective way of doing it as well. And there are other ways. You can look byte by byte. There are a variety of other techniques. Anyway, but this is a cute one.

Here's another one, population count. Count up the number of one bits in a word. So here's one way of doing it. I start out r at zero. And I keep incrementing r. And what I do is I quit when x is zero. And what I do is I do that trick of x, ending it with x minus 1. Which does what? Eliminates the low order bit, that's one, the low order one.

So basically I go through and I just kick out one of the ones, kick out another one of the ones, kick out another one of the ones, until I'm done. This has a branch in it. But in some sense it's a predictable branch because almost all the time you're going through the loop.

However, it has downside, which is that suppose you're given minus 1. Then you have to do 64 iterations of this loop before you can get your final answer. And so that's a lot of iterations to do. So here's what's going on in your loop. Here's x. Here's x minus 1. And now if I and them it's very similar to the other trick that I taught you. You and them, notice you have the same number you started with except it's missing the one low order bit.

So this is fast if the population count is small. If you know there's only a couple of its on in the word, then this can be a pretty effective technique. But in the worst case it's going to take it's going to be proportional to the number of bits in the word. Because you're only getting rid of one bit at a time. But it's better, in some sense, than looking one bit at a time because you have the off chance that the number of one bits will be sparse. Whereas if you just looked at the low order bit, then the next bit, then the next bit, that would definitely take you worst case every single time.

Here's another way to do it. It's a table look up. So you have to pay, but if you're doing this a lot maybe all of this is an L1 so the table look up only costs you four cycles if it's an L1 cache. So what is this sequence? This tells for any given byte, so there's 256 values, how many ones are in the world. So zero has zero one bits. One has one one bit. Two has one one bit, three has two one bits. Four has one one bit, five has two, six has two, seven has three, eight has one, et cetera. So that's this table. I didn't fill out the rest of the table.

And now what you're doing in this loop is you're basically taking a look at the value of x. You're right shifting it and then your indexing, masking with the low order byte. So you may ask the low order byte. You add that to the count by doing a look up, which hopefully only takes you four cycles if the table is in L1. And then just run around this loop until you've got no more things in your word. So how many are in byte one, how many are in byte two, how many are in byte three, and so forth.

For things that use table look up you have to be careful because if you have a great big table why not look up two bytes at a time? Well two bytes is 65,000 entries. So why not look up four bytes at a time? Four bytes is four billion entries. At that point you're going out to memory and starting to consume a lot of space.

So here's some common numbers. These are sort of approximate. But generally if you're doing operations on registers, one cycles, and plus you can issue six per core, per cycle. So L1 cache is going to cost you around four cycles, L2 cache about 10, L3 about 50, and D RAM about 150 to 200 cycles.

When you access these you get, in fact, generally for all these, you tend to get a 64 byte cache line. So you're getting more than one. But if what you're doing is random access in a table it doesn't help that all those other bytes are coming in.

Population count three using parallel, divide, and conquer. Here's the clever one. Here's the code. It's all register operations basically.

So it's creating some masks. So let's just take a look at what does this first instruction do? It's taking minus one. It's shifting it left 32 bits. So that gives all ones in the higher order half of the word, and all zeros in the lower half. And then it's xor-ing it with minus one. But with all minus ones.

So that gives you a mask of ones in the low order half of the word. Yeah question?

AUDIENCE: Why don't you just do negative one right shifted. Isn't there a type of-

PROFESSOR: Yeah you can do that. I was trying to be consistent here. And, in fact, for these first two operations there are actually more clever ways of doing this that take fewer operations.

AUDIENCE: Isn't there a right shift operator that [? pulls some ?] zeros in the top level?

PROFESSOR: Yeah, so there's logical versus arithmetic right shift. Yeah.

AUDIENCE: [INAUDIBLE]

PROFESSOR: That's right. But then I wouldn't have the pattern that I'm setting up here. So yes, in fact, if you want to play with it yourself you can optimize these first two statements. They don't need to be as complicated as this one. But basically what you're doing in every step is your shifting it over, half the word, xor-ing it. And then the second one is you get a block of 16 bits of zeros, 16 ones, 16 zeros, 16 ones. The next one you get a block of eight zeros, eight one's, eight zeros, eight ones.

And so basically you're generating masks for that. So by the time you get down to the last one you're having every other bit is zero. You're alternating zeros and ones.

And then, basically- well let me not go through the code here. Let me show with an example. The main thing to observe is that it takes log n time where n is the word length to do this. So here's population count on 32 bits, same kind of thing.

So here's the idea. We extract every other bit for the two words. So you saw how I extracted that right? So we extract. So I can do that with a mask and a shift.

And then I add them together. So just so we can see what's being added here. And when I add them together the largest value I'm going to have in any one of these things is what?

AUDIENCE: Two.

PROFESSOR: And [? that, ?] fortunately, fits in two bits. So we can get off the ground. So now every two bits has the sum of the two bits that were there. The bits that I'm not showing are all zeros because it's done 64 bit words. The bits I'm showing are all zeros.

So now what do we do? We mask and shift and take off every two pairs of bits and then add them together. So now this guy is saying there's four bit that were originally in the word that began. This one says there are two bits in that range, one bit, one bit, two bits, three bits, two bits, two bits.

So we do it again. Add it together. And we just keep going. And then finally we add them all together. It says there are 17 ones in the word, which there were. I should have probably left the word up there or something so we could verify that. But Yeah there are 17 ones.

So everybody see it's parallel, divide, and conquer because you're adding many words, many sub-pieces pieces of the word [UNINTELLIGIBLE]. And the key thing is to make it so that no carries are propagating out of their range. But the numbers are just getting smaller and smaller. When you're done you're only going to have six bits here that are significant anyway. All of these will be zeros. Is that cool? So there's a 17, yeah.

Here's a problem for which bit representations are a lot of fun. Last year we gave this as a problem to students. This year we're giving you a different problem so that lets me lecture on it. So many people are probably familiar with this problem. It's an old chess nut.

But basically the queen's problems is to place n queens on an n by n chess board so that no queen attacks another. So there are no two Queens in any row, column, or diagonal. So queen's kind of move like this. It's got to be clear. If I did that around any one of these guys they wouldn't hit anybody else.

In fact, this arrangement here is, I think, one of the few symmetric ones. Maybe it's the only symmetric one. It's radially symmetric. Most of them are more scattered.

So the question is how do you find such a thing, or count the number of solutions, is another popular one, et cetera. A popular strategy for this is called backtracking search. And we're going to have in your homework a different backtracking search. And the idea is you just simply try to place the queens row by row.

So, for example, we start out with the first row, row zero, and we place a queen. Then we go to the next row. And we try to see if a queen works on that square, nope, nope, yes it works there. Now we go on to the next row.

So this is making progress. We're placing queens. We're going to more rows. We're going to get to the end of the rows, right? So we keep- yep, we found it. And we keep going. This is easy. Found it right after two there. That's pretty good. Found it there. Look, we're making great progress. Doesn't go there, doesn't go there, doesn't go there, doesn't go there, oops doesn't go anywhere.

So what do we do? We backtrack. We say, gee, if it didn't fit in any those but it had nothing to do with the placement there, it's the fault of the guy who came before me. So that position is not a valid position, at least with that prefix. So we continue with him. Aha, we found another place for him.

So then we try this one. Oops this doesn't look good, aha, got to backtrack. Whoops that's the last one in the row, got to backtrack again. So that means that guy can't go there. Found a place, now we get to go forward, hooray. And you keep going on. So you backtrack, et cetera, until you finally find a place for them.

So the backtracking search is pretty interesting. But the question is how do you represent it so that it can go really fast. So here's some ideas. The first idea you might come up with is to use an array of n squared bytes, where you put a value in the byte if there's a queen there.

You should, at this point, figure out that, gee all I have to know is whether a queen is there or not. So why should I keep a byte? Why not just keep a bit? That'll be smaller and have a smaller representing.

So let's keep n squared bits. Well let's see. If I'm only putting a queen in one place in every row I never have to have more than one bit set in any row. So why not just say the column number in the row that I'm in? So rather than using an array of n bits for every row let me just use an index of a byte to say which row that one queen is in because there can't be any other Queens in that row in a legal configuration. So that's actually more clever. And that's the way most people code it.

But we're going to look at a solution that was originally due to Edsger Dijkstra of using three bit vectors to represent the board. And the idea is we want to make things go- we're going to use three bit vectors that are relatively small.

So it turns out the n queens is an exponential search problem. And so you really can't run n queens on 128 by 128 board. You can run that if you're interested in one solution. You can't count up how many solutions. And if you go to Wikipedia and look at n Queens they will tell you what all the latest records are for who has computed how many solutions there are on an n by n board for n up to some rather small number. So it's a way to get your name on the web, which, as you know, is very difficult to do.

So let's see this three bit vector trick. So the idea of the bit vector trick is that, for any partial placement, rather than representing where the queens are on the board, what I really care about is which columns have been knocked out.

So, therefore, what I'll do is I'll store a one if there's a queen in that column and a zero if I don't have a queen in that column. So the point is I can keep the whole representation of the column mask in a word that just tells me whether I have ones or zeros in a column. Now how do I know whether it's safe to place a queen in a given column? What's that?

AUDIENCE: [INAUDIBLE]

PROFESSOR: How do I know, if I try to place a queen say, here, how do I know whether that's OK or not? Suppose that my program wants to try to put a queen in this space. It can't because there's something here. It's just some operations on words, which we will see here.

So placing a queen in column c is not safe if this down were here. When I and it with one shifted left by the number of columns so that I have the position of the queen in the column if that's non-zero because it means somebody else is in that column. If it's in a new column when I do the and I'll get all zeros. Everybody follow that?

So testing columnists, whether it's safe to put it in a given column, from the column attack we can do that really, really efficiently, right? These are all going to be in registers. No memory operations, no table look ups, no L1 caches, all right in registers.

Well what do we do about the diagonals? So for the diagonals we can also use a bit vector representation for each diagonal. Where I look to see if there's a number along this diagonal, and if there is a queen then it's a one. And if there isn't a queen on the diagonal it's zero. There are more diagonals then there are columns, right? So I have a longer bit vector representation. I have to represent that. But I can still do that in one computer word for things of different size, or even for things that are pretty good size two computer words would be, certainly, ample.

So now how do I tell whether or not a queen placed on a given square can legally be placed there? So it turns out it's not safe to place it if when I take my writer ray, and I take n minus r plus c, and left shift it by that amount, and that's non-zero. So here I'm indexing rows and columns from the upper left hand corner. And so, basically, you're trying to say is, for a given square, notice that if I increase the row, that's this way, I get to the same- of if I'm increasing the row, I'm decreasing the diagonal that I'm on. But if I'm increasing the column I'm increasing the diagonal I'm on.

So that's it's basically a difference here. And then you just normalize it and with essentially very, very few operations I can tell whether there's a conflict in a column. Of course, for both this and the other one, if I need to set it now that's pretty simple also. I just take this and or it with right. If my test is good, I just take this, or it with right, and now that's my new right.

And left is similar. So, once again, we have these guys going this way, placing a queen in row r. And column c is not safe if- and now I just look at row plus column because these diagonals increase with both row and column. You're increasing the diagonal for both row and column.

And so I'm not going to go through all the details of the code. But you can see that with this representation, literally, the inner loop of your program, which is testing whether queens fit on boards and then setting them if they do, you can do with just three words and a few operations. Question?

AUDIENCE: Can you mask the three bits together and check if one position is taken up? You'd have to do some creative stuff with lining up the row and column vectors with the diagonals, but-

PROFESSOR: So typically here you're looking at both row and column. So you're adding them, whereas on the previous one you were subtracting. And then the first one you didn't even care about what the column was. So I'm not sure I would know how to combine those. It's conceivable you could do it.

AUDIENCE: [INAUDIBLE] column vector, you could add the two together and find specific positions that are free.

PROFESSOR: Yeah so you could also do a generation that says here are the ones that are free and then use things like the least significant bit trick to pull out what is the positions I should bother to check. So here I can test that it's safe. But I could also generate using similar tricks, which is what you're saying, generate all of the bit positions on a given row where it would be safe to put a queen. Yep, yep, good.

So fast programs use this technique. So you see that there's a lot of cleverness in these kinds of techniques. So there are a whole bunch of other bit hacking techniques. One really good resource for it is this webpage. And of course I'll put this up on the- called bit twiddling hacks, where he's compiled- there's a lot of people that have worked on different bit twiddling hacks. And he's done a very good job of compiling what he thinks are the best code sequences for a whole bunch of things, including things like reversing the bits in a word. If you think about it that could be kind of tricky. Actually turns out to be relevant to your homework as well.

So on your homework, so lab one will be- it's posted I gather, right? It'll be posted shortly. We have you trying to take advantage of some of these bit tricks in a couple of warm up exercises and then a backtracking search algorithm. And so I think you'll find it's a lot of fun. You'll learn a lot about all the kinds of tricks you can do to make stuff go fast by using register operations locally, and using good representations for your storage.

Yes, also, announcement. Tonight, at 7 o'clock, in 32, 144, there is a primer on c. So if you want to brush up on your c or if you want to learn c, this is a good time to go. There's a lot of good nuggets of wisdom coming out for c. OK, thanks very much. See you Thursday.