JULIAN SHUN: Good afternoon, everybody. So welcome to the third lecture of 6.172. Today we're going to talk about bit hacks, and today's going to be a really fun lecture. So, first of all, let's recall the binary representation of a word. So a w-bit word is represented as follows. So we're going to number the bits from x0 to xw minus 1 starting from the rightmost side.

And the unsigned integer value stored in x with this binary representation can be computed as follows. So it's essentially the sum of a whole bunch of powers of 2. And you sum the product of the bit with the appropriate power of 2. So if the bit is 1 in position k, then you multiply by 2 to the k. And if it's 0, then you just add 0.

So, for example, let's say we have this 8-bit word here. And if we apply this equation, we get-- first we get 2 because there is one bit in the first position. So we multiply 1 by 2 to 1, which is 2. Then in the second position, we also have a 1. So we multiply 1 by 2 to the 2, which is 4. And then we have 16 and 128. So we just sum up all of these powers of 2 and that gives us the unsigned integer value.

And that 0b prefix here represents a Boolean constant. So that means we're going to interpret this constant as a Boolean value. There's also signed integers. So you can also represent negative numbers, which is useful, and this is called the two's complement representation. And here's the formula for computing the two's complement representation of a word.

So for bit 0 all the way up to bit w minus 2, you do the same thing as above. But for the leftmost bit or bit w minus 1, you subtract that bit multiplied by 2 to the w minus 1. So for this example here, we saw 2 plus 4 plus 16. That's the same as above. But for the leftmost bit, since we have a 1 here, we're going to subtract 2 the 7, which is 128. And this gives us the signed value for the integer, which is negative 106.

Does that make sense? Any questions about this representation? So the leftmost bit is known as a sign bit because it tells you whether you need to subtract by this negative value or not. So if it's 0, then you don't have to subtract anything. If it's 1, then you subtract by a large integer value.

So in two's complement, the all 0's word is just 0. So you just apply the above formula and everything is 0. So you just get 0. What's the value of the all 1's word? Yes.

AUDIENCE: 1.

JULIAN SHUN: Negative 1, right? So the reason why it's negative 1, so you can just use the formula. And we're going to sum up a bunch of powers of 2. All of the x sub k's are going to be 1. So we're summing up 2 to the k from k equals 0 to w minus 2, and that's a geometric series which sums to 2 to the w minus 1 minus 1. And then for the sign bit, we're going to subtract 2 to the w minus 1. So now the 2 to the w minus 1's cancel out and we're just left with negative 1.

So this is an important property to know about two's complement representation. The all 1's word is just negative 1. And this leads to important identity which says that x plus the one's complement of x-- the one's complement is just all the bits of x flipped-- is equal to negative 1. This is because if you add x with all of it bits flipped, then you're just going to end up with the all 1's word. And we saw on the previous slide that that's equal to negative 1.

And from this identity, we have that negative x is equal to the one's complement of x plus 1. So this relates the two's complement to the one's complement representation. Let's look at an example. So let's look at-- let's say x is equal to this constant here. The one's complement of x or tilde of x is just all of the bits of x flipped.

And then to get negative x, we add 1 to the one's complement of x. And the fact of adding 1 here is we're going to take the rightmost 0 bit in the one's complement, flip that to a 1. And then for all of the bits to the right of that, we flip them to 0's. So another way to see this is you look at the representation of x and you flip all of the bits up to the rightmost 1 but not including that rightmost 1 bit, and then you just copy everything over.

So any questions about this? OK. So this is a table showing the relationship between hex and binary representation. So hex representation is base 16. And the reason why we use hex is because sometimes we have these big binary constants and we don't want to write-- have to type all of these symbols into our code. And hex gives us a more compact format to write these constants.

And this table, you can basically just look up, for each possible hex value, what its binary representation is. And for the values from 0 to 9, we're just going to use the same as decimal representation for hex. And then for values 10 to 15, we're going to use the characters from A to F.

To translate from hex to binary, you just take each hex digit, look it up in this table, write out the binary equivalent, and then you concatenate together all of the binary values you've got. So in this example I have this hex constant which says DEC1DE2C0DE4F00D. So now I just look up each of these hex values in this table. So D is 1101, E is 1110, C is 1100, and so on. And I just concatenate all of these values together and that gives me my binary representation.

And you can also go the other way around, converting binary to hex. And you do the same thing, just look it up in this table. And the prefix 0x here designates a hex constant, just like 0b designates the Boolean constant. So if you're using these constants in your code and you're writing it in hex, then you should use the 0x prefix.

So C has a bunch of bitwise operators. And here's a table describing what these bitwise operators do. So the ampersand is just logical AND. The vertical bar is logical OR. This caret sign is the XOR or exclusive OR. And XOR just says that if either of the two bits is 1, then we return 1. And if both of the bits are 0 or both of them are 1, then we return 0.

The tilde sign is the one's complement or the not. And then we have left shift and right shift operators. So let's look at how these operatives work on this example here. So we have these two 8-bit words, A and B. To compute A AND B, we just look at every two bits in the same position in A and B and compute the AND of those two bits. So 1 ANDed with 0 is 0, so we get 0 here. 0 ANDed with 1 is 0. 1 ended with 1 is 1, and so on.

A OR B is similar but now you apply the OR operator instead of the AND operator. So if either one of the two positions is 1, then you return 1. And if both are 0, then you return 0. So an A OR B, all of the bits except for this bit here is 0 because in the original two words both of the corresponding bits were 0.

For A XOR B, we check if exactly one of the two bits is 1. So for the leftmost bit, we have 1 and 0, so we have exactly one bit set to 1 and we get a 1 here. The second bit is 0 and 1, so that's 1. The third bit is 1, 1, so that's 0, and so on. Tilde of A is just the one's complement of A. We saw that before. We just flip all the bits.

A right shifted by 3, we just shift the bit string to the right by 3, and then we fill in the digits or the bits on the left with 0's. And then A left shifted with 2, we do the same thing but to the left. And then we fill in these empty bits with 0's. So these are the bitwise operators in C. Any questions?

AUDIENCE: They're not [INAUDIBLE]?

JULIAN SHUN: For a right shift, there is a-- there is a shift that will fill in the upper digits with whatever the leftmost digit was. But if you're working with unsigned integers, then it's not going to do that. For signed integers it will. And when we're doing bit manipulations, we're usually going to stick to unsigned integers, so we don't have to worry about that.

So now let's look at some common idioms that you can do using these bitwise operators. So the first one we'll look at is setting the kth bit in a word x to 1. So the idea here is to use a shift followed by an OR. So we're going to compute 1 left-shift it by k if we want to set the kth bit to a 1. And this gives us a mask with a 1 in exactly the kth position, and 0's everywhere else.

And then now when we OR that in to x, that's going to change the bit from a 0 to a 1 if it was a 0. And if that bit was already set to 1, then this doesn't do anything. And then for all of the other positions, since we're doing an OR with 0, we're just copying over the bits from x. So that's setting the kth bit.

We can also clear the kth bit. And the idea here is to use a shift, a complement, and then an AND. So again we're going to generate this mask, 1 left-shifted by k. But now we're going to take the complement of this. So now we have a 0 in exactly the kth position and 1's everywhere else.

And now when we AND this mask with x, in the kth position it's going to clear that bit because we're ANDing it with a 0. So the result is going to be 0 no matter what was there before. And then for all the remaining bits, since we're ANDing with 1, we're just copying it over from the original word.

You can toggle the kth bit or flip the kth bit using a shift and then an XOR. So, again, we're going to generate this mask. And then now, when we do an XOR with this mask, it's going to change a bit from a 0 to 1, or from a 1 to a 0, because that's what XOR does. So in this example, it's changing from a 0 to 1. But if it was already a 1, then it's going to toggle it back to 0. Any questions?

So let's look at another bit trick. So here we're trying to extract a bit field from a word x. And this is important if you're working with encoded data. And the idea here is to do a mask and a shift. So we're going to generate a mask with 1's in exactly the positions that we want to extract out of this word, and then 0's everywhere else. And then we're going to AND the x with the mask, and that's going to give us the bits in the four positions that we wanted to extract in this example, and then we have 0's everywhere else.

And then now we're going to right-shift this value that we extracted so that it appears in the least significant digits so that we can use it in our computation. So this is a very useful bit trick to know if you're working with compressed or encoded data. And if you use the bit field facilities in C, it's actually going to generate assembly code that will do masking and shifting for you.

You can also set a bit field in a word. So let's say we want to set a bit field in x to some value y. The idea is to first invert this mask to clear those bits we want to set in x. And then we OR in the shifted value of y. So let's say we have these two words, x and y here. We're going to generate the mask as we did before, but now we're going to flip all the bits in the mask by taking the one's complement.

And then we AND the-- we AND the one's complement of the mask with x, and that's going to clear the bits in x because we have 0's in exactly those positions in that mask, and when you AND that into x it will return to 0. And then for all the other positions, we're just copying in the bits of x.

And then, finally, we're going to left-shift y by an appropriate amount so that we can line up the value with these four bit positions here. And then we can now just OR those values in. And this will set the positions in x to the value y.

In order to be safe, you should actually do a mask on the shifted y value before you OR it in, because you don't know that the value of y is within the range of the mask. So if y has some garbage values in the higher bits, when you OR this in it might pollute the original value of x. So, for safety, you should actually do a mask before you OR the value, the shifted value of y in. So any questions on this?

So now let's look at how we can swap two integers. So we want to swap the values of x and y. The standard way to do this is to use a temporary variable t. So we set t equal to x, x equal to y, and then y equal to t. This does involve a temporary variable, however. So now the question is whether we can do a swap without using a temporary variable. It turns out that you can using bit tricks.

So here's the code for doing a no-temp swap. So you first set x equal to x XOR y, then y equal to x XOR y, and then x equal to x XOR y. So has anyone seen this before? OK, good. So some of you have seen this before. And for the rest of you all, I'll tell you how it works in the next couple slides.

So let's first look at an example of how to run this code before we go into why it actually works. So we're going to start with these two words in x and y. We're first going to do x equal x XOR y. And now we store the result in x. And this is the result when you do the XOR of these two words.

And then now we do y equal to x XOR y. And notice how the value of x here has already changed. So we're doing the XOR of these two words and setting that to y. And here this value is actually the same as x. So we've already placed x in y.

And, finally, we do another XOR. We set x equal to x XOR y. And then this gives us this value, which is y. So at the end, we've just swapped x and y without using any temporary variable.

So the reason why this works is because XOR is its own inverse. So if you do x XOR y, and then XOR the result of that with y, you just get back x itself. So let's look at the truth table to see why this is true. So in the x and y columns, I've shown all the possibilities. So there are four different possibilities of x and y. And then I also have the values of x XOR y. So it's 1 in the rows where I have exactly one 1, and then 0 in the remaining rows.

And then now if I do x XOR y XORed with y, I'm going to XOR these values with y. 0 XOR 0 is 0. 1 XOR 1 is 0. 1 XOR 0 is 1. And 0 XOR 1 is 1. And notice that these values are the same as the values of x. So when I XOR something in twice, it just cancels out and I get back the original thing.

So now let's go into why this bit trick actually does a swap. So in the first line, what we're doing is we're generating a mask with 1's where the bits in x and y differ. Because that's what XOR is going to give you. It's going to return a 1 if the bits are different, and 0 otherwise. So this is a mask that tells us in which positions the bits in x and y differ. And I'm going to store that into x.

And then in the second line, when I do x XOR y, this is going to flip the bits in y that are different from x, because I'm XORing with this mask, which tells me which of the bits differ from x. And then if I XOR with that mask, I'm flipping the bits in y that differ from x, and this will just give me back x. And I store that in y. So we see that the original value of x is in y now.

And then in the last line, I do the same thing but now I'm flipping the bits in x that are different from y. So I still have the mask that's stored in x. And then I can XOR that mask with y, and y has the original value of x. So this is flipping the bits in x that differ from y, and now I have the original value of y stored in x. So this is a pretty cool trick, right? Any questions on why this works?

So one thing about this bit trick here is that it's actually poor at exploiting instruction-level parallelism, so it's actually going to be slower than the naive code that uses a temporary variable. Because in the original code I had, I could actually execute two lines in parallel. I can store value into the temporary and then also change one of the values of x and y at the same time. Whereas in this code here, there's a sequential dependence among these three lines. I can't execute any of the lines in parallel.

We'll learn more about instruction-level parallelism in next week's lectures, but I just wanted to point out that the performance of this isn't actually that great. But this is actually a pretty cool trick to know. Sometimes it shows up in job interviews.

So the next thing we're going to look at is finding the minimum of two integers, x and y. So let's say we want to store the result of the minimum in a variable r. Here's the standard way to do this. We just use an if-else statement. So if x is less than y, than r is x. And, otherwise, r is set to y.

Here's an equivalent expression. It just uses the ternary operator in C. It does exactly the same thing as the if-else statement on the left. One performance problem with this code is that there is a branch in the code. So we have this if statement that checks if x is less than y. And modern machines will do branch prediction. And for whatever branch it predicts the code to take, it's going to do prefetching and execute some of the instructions in advance.

But the problem is if it mispredicts the branch, it does a lot of wasted work, and the processor has to empty the pipeline and undo all of the work that it did. So this is a performance issue due to branch misprediction. Modern compilers are usually good enough to optimize this branch away, but sometimes the compiler isn't good enough to optimize the branch away.

So is there a way to do a minimum without using a branch? All right. So here's how you do it. So we set r equal to y XOR x or y ANDed with negative x less than y. So it's pretty obvious, right? So why does this work?

So first we need to know that the C language represents the Boolean values true and false with the integers 1 and 0, respectively. So now let's look at the two possible cases. First, let's look at a case where x is less than y, and then we'll look at the case where x is greater than or equal to y.

So in the first case, when x is less than y, the comparison here x less than y is going to return 1. And then we're going to negate that, which gives us negative 1. And recall from earlier, negative 1 is the all 1's word in two's complement representation. So when we AND x XOR y with all 1's word, that just gives us x XOR y.

And now we're left with y XOR x XOR y. And we know that the-- we know that the inverse of XOR is itself. And therefore the two y's cancel out here and we're just left with x. And in this case x is indeed the minimum.

In the other case, we have x greater than or equal to y. Then the expression x less than y is going to return 0. Negative of 0 is still 0. And then when we AND x XOR y with 0, we're left with 0. And this just gives us y XOR 0, which is y. And in this case y is the minimum of the two integers. So any questions?

So how many of you-- how many of you knew this already? Good. So we learned something new today. So let's see how branches work in a real function. So here we're trying to merge together two sorted arrays, and this is a subroutine that's used in merge sort if you've seen it before.

So the inputs to this function are three arrays. So we want to merge together arrays A and B and store the result in C. And then we also pass the function the sizes of A and B in na and nb. So what does the restrict keyword do here? Does anyone know?

So the restrict keyword tells the compiler that this is going to be the only pointer that can point to that particular data. And this enables the compiler to do more optimizations. So when you're writing programs and you know that there can only be one pointer pointing to specific pieces of data, then you can declare that restrict keyword, and this gives the compiler more freedom to do optimizations.

So now let's look at this procedure here. So while the sizes of A and B are nonzero, we're going to go into this if-else clause and we're going to check if the element pointed to by A is less than or equal to the element pointed to by B. And if so, we're going to store that element pointed to by A into C. And then we're going to increment both the C and A pointers. And then we're going to decrement na. This tells us that there's one less element in A that we need to merge in now.

And, otherwise, we do the same thing but with array B and nb. And if one of the two arrays becomes empty, then we go to one of these two while loops at the bottom and we just copy all the remaining elements in the non-empty array into C. So here, if na is greater than 0, then A is a non-empty array, and then we just copy the remaining elements of A into C. And, otherwise, we copy the remaining elements of B into C.

So let's do a simple example. Let's say we want to merge these two arrays in green into the blue array here. So let's say the top array is A, and the bottom array is B, and the blue array is C. So, initially, A and B are pointing to the beginning of these two green arrays.

And since both arrays are non-empty, we're going to compare the first two elements here. And we see that 3 is less than 4, so we're going to place 3 into the array C. And then we're going to increment the pointer in A to point to the next element. And we're also going to increment the pointer C to point to the next slot.

Now we're going to compare 4 and 12. 4 is less than 12, so we place 4 into the array C, and we increment array B. And then we just keep doing this. So 12 is less than 14. 14 is less than 19. 19 Is less than 21. 21 is less than 46. And here 23 is less than 46.

And at this point, one of the arrays becomes empty. So B is empty now. So now we get to the second while loop. And we see that A still has elements in it, and we just copy the remaining elements in A into C. And then we're done. So that's how the standard code for merging two sorted arrays works.

So let's look at each of these branches to see if it's predictable. So a predictable branch is a branch that most of the time it returns the same answer, and only rarely does it return a different answer. And an unpredictable branch is one where it sometimes returns one value and sometimes returns another value and you can't really predict it. So let's look at the first branch. Does anyone know if this branch is predictable? Yes.

AUDIENCE: That would be unpredictable because it depends on what input you're given.

JULIAN SHUN: So it turns out that this branch is actually predictable because it's going to return true most of the time except for the last time. So it's only going to return false when nb is equal to 0. And at that point you're just going to execute this once and then you're done. But most of the time nb is going to be greater than 0 when you execute this, and we call this a predictable branch. What about the second one? So--

AUDIENCE: Also predictable?

JULIAN SHUN: Yes. So it's also predictable for the same reason. What about the third one? Yes.

AUDIENCE: No. Because we really-- if we already knew which was bigger, then we already have the sorted array then.

JULIAN SHUN: Yes. So this turns out to be unpredictable because we don't know the values in A and B a priori. So this condition inside the if statement is going to return true about half of the time because we don't know what values are in A and B. And that's going to be an unpredictable branch because it's going to return true or false about 50/50.

What about the last one? Yes. Why?

AUDIENCE: Yes, because for similar reasons of 1 and 2. It's probably [INAUDIBLE].

JULIAN SHUN: Yes. So it is predictable. The reason why it's predictable is that most the time it's going to return true. And that once it returns false you're never going to look at that again inside this function call. So it returns true most of the time, and we call that a predictable branch.

So branches 1, 2, and 4 are OK because they're predictable branches, but branch 3 is going to cause a problem. It's an unpredictable branch, and the hardware doesn't really like this because it can't do prefetching efficiently. So to fix this, we can use our no-branch minimum bit trick that we learned a couple slides ago.

So now what we're doing is we're going to have a variable called cmp which stores the result of the comparison between the first element of A and the first element of B. And then now we're going to get the minimum of A and B as follows. It's the same bit trick that we saw before. So now the variable min is going to store the smaller of the first element of A and the first element of B. And we also have the result of this comparison here. So that's stored in cmp.

So first we're going to place the minimum value in C. And then, based on the result of cmp, we're going to increment one of A or B. So if A was less than or equal to B, then cmp is going to be 1. And A plus equal cmp is going to increment A by 1. And then B plus equal to not cmp is going to not do anything, because not cmp is 0.

And then for na, we're going to decrement by cmp. So it's going to be 1 if A is less than or equal to B, and 0 otherwise. And then for nb, we're going to decrement by the not of the cmp. So only one of these two lines is actually going to do something based on the result of the comparison. And then the rest of the code is the same as before. Any questions? So now we've gotten rid of this unpredictable branch that we had before.

So one thing about this optimization is that it works well on certain machines. However, on modern machines, using a good compiler like Clang with the minus O3 flag, the branchless version is usually going to be slower than the branching version because the compiler is actually smart enough to get rid of the branch inside the original version of minimum. There's this instruction called cmov or a conditional move. It's basically a branchless instruction for doing a comparison. We'll learn more about that next week.

So this trick actually usually doesn't really work. There might be some machines and some compilers that works, but most of the time, the compiler is better at optimizing this code than you are. So one of the common themes so far is that I've told you about a really cool bit trick and then I told you that it doesn't really work. So why are we even learning about these bit tricks then if they don't even work?

So first is because the compiler does some of these bit tricks, and it's helpful to understand what these bit tricks are so you can figure out what the compiler is doing when you look at the assembly code. Secondly, sometimes the compiler doesn't do these optimizations for you and you have to do it yourself. Thirdly, many bit hacks for words extend naturally to bit and word hacks for vectors, which are widely used in high-performance code. So it's good to know about these tricks. These bit tricks also arise in other domains. And, finally, because they're just fun to learn about. And for project 1, you'll be playing around with some of these bit tricks, so it's good to know about these things that I've talked about already.

Here I'll talk about a bit trick that actually does work. So here we're trying to do modular addition. So we want to do x plus y mod n. And here let's assume that x is between 0 and n minus 1, and y is also between 0 and n minus 1. So the standard way to do this is just to use the mod operator, x plus y mod n. However, this does a division, which is relatively expensive compared to other operations unless n is a power of 2. But most of the time, you don't know if n is a power of 2 at compile time, so the compiler can't actually translate this to a right shift operation, and then it has to do a division.

So here's another way to do it without using division. So we're first going to set z equal to the sum of x and y. And then if z is less than n, then it's already within the range and we can just return z. If z is greater than or equal to n, well we know we can be at most 2n minus 2 because x and y were both at most n minus 1. So all we have to do is to subtract n and bring it back into range.

However, this code has an unpredictable branch here because we don't know whether z is less than n or not. So now we can use the same trick as minimum. So now we're going to set r equal to z minus n ANDed with the negative of z greater than or equal to n.

So if z is less than n, then this is going to return 0 in here. And n ANDed with 0 is 0, so we're just left with z. And if z is greater than or equal to n, then this is going to be 1. We negate that, we get negative 1, which is the all 1's word. n ANDed with all 1's is just n. So that is z minus n, which will bring the result back into range.

So any questions? Yes.

AUDIENCE: It seems like there essentially is still a branch based on the value of z. So why would that be faster?

JULIAN SHUN: So this branch here is just generating either a Boolean value 1 or 0. There's actually-- like the code that you execute after it, it's still the same in either case. So the branch misprediction only hurts you if there are two different code paths. In this version, there are two different code paths, because one is doing z and one is doing z minus n.

So the next problem we will look at is computing or rounding a value up to the nearest power of 2. And this is just 2 to the ceiling of log base 2 of n. And recall that lg of n is the log base 2 of n. That's the notation we'll be using in this class. Here's some code to do this.

So we have our value of n here. First, we're going to decrement n. And then we're going to do an OR of n with n right-shifted by 1. Then an OR with n and n right-shifted by 2, and so on, all the way up to 32. So we do this for all powers of 2 up to 32. And then, finally, we increment n at the end. So let's look at an example to see why this works.

So we're starting with this value of n here. First we're going to decrement it. And what that does is it flips the rightmost 1 bit to 0, and then it fills in all the 0's right of that with 1's. And then when we do this line, which says n is equal to n ORed with n right-shifted by 1, that's essentially propagating all of the 1 bits one position to the right and then ORing those in.

So we can see that this 1 bit got copied one position to the right. This 1 bit got copied to one position to the right. These 1's also propagate, but since they were already 1's it doesn't do anything.

For the next line, we're propagating the 1 bits two positions to the right. So this 1 bit here gets copied here. This 1 gets copied here, and so on. And then the next line is going to propagate bits four positions the right. Then 8, 16, and 32. For this example here, when I get to this line I'm already done. But, in general, you have more bits in a word, which I can't fit on this slide.

And now we have something that's exactly one less than a power of 2. And when we add 1 to that, we just get a power of 2. So we're going to zero out all of these 1 bits and then place a 1 here. And this is exactly the power of 2 that's greater than the value n.

So the first line here is essentially guaranteeing us that the log nth minus 1 bit is set. And we need that bit to be set because we want to propagate that bit to all the positions to the right of it. And then these six lines here are populating all the bits to the right with 1's. And then the last bit is setting the log nth bit to 1 and then clearing all of the other bits.

So one question is why did we have to decrement n at the beginning? Yes.

AUDIENCE: In case n is already [INAUDIBLE].

JULIAN SHUN: Yes. So if n is already a power of 2 and if we don't decrement n, this is isn't going to work because the log nth minus 1 bit isn't set. But if we decrement n, then it's going to guarantee us that the log nth minus 1 bit is set so that we can propagate that to the right. Any questions? Yes.

AUDIENCE: [INAUDIBLE]?

JULIAN SHUN: Because, in general, you're using 64-bit words. Here I don't have that many bits here because

I can't fit in on the slide, but in general you have more bits. Let's look at another problem. Here we want to compute the mask of the least significant 1 in a word x. So we want a mask that has a 1 in only the position of the least significant 1 in x, and 0's everywhere else. So how can we do this?

So we can set r, the result, equal to x ANDed with negative x. So let's look at why this works. So here is x. And recall negative x is the two's complement of x plus 1. So what we do is we flip all of the bits up to the rightmost 1 but not including it, and then we just copy all of the bits over. That's how we get negative x from x.

And then now when we compare x and negative x, we see that all of the bits when we AND them together are going to be 0 except for the bit at the position corresponding to the least significant 1 bit in x. And that's going to be 1 since we're ANDing 1 and 1, and everything else is going to be 0. And this will give us the mask that we want. So this works because the binary representation of minus x is just the one's complement of x plus 1.

So now, a question is how can we find the index of this bit? So here I'm just generating a mask that has a 1 in the least significant 1 in x, but it doesn't actually tell me the index of this bit. In other words, I want to find the log base 2 of a power of 2. So that's the problem we want to solve, and here's some code that lets us do this.

So we have this constant called the de Bruijn. It's written in hex here. And then we have this table of size 64 called convert. And now all we have to do is multiply x by this de Bruijn constant, right shift it by 58 positions, and then look up the result in the convert table. And that's going to give us the log base 2 of the power of 2. Any questions?

[STUDENTS LAUGH]

So this looks like magic to us. So in the spirit of magic, we're going to do a mathemagic trick. And to do this trick, I'm going to need five volunteers, and the only requirement is that you need to be able to follow directions. So who wants to volunteer for this magic trick? Yes, 1, 2, 3, 4-- one more-- 5. All right, come on up. So line up here.

[STUDENTS APPLAUD]

Yes, just line up right here. Can you move a little bit over to the left? OK, cool. So today I have the pleasure of welcoming Jess Ray, also known as The Golden Raytio, to join us for a lecture and help us perform this cool magic trick. So let's give her a round of applause.

[STUDENTS APPLAUD]

JESS RAY: I'm going to be doing a little bit of magic trick for you all today. I'm going to be reading your guys' minds. And I know you're looking skeptical, but I'm hoping I can convince you here. So we'll get to that part in a second. But, first, the first big step in reading minds is you got to clear the air, like get rid of all the negative vibes, all the bad energy. Throw that out. So I'm going to need a little help from you guys in doing this.

So, first, we have this sweet little bell here. Let's see. Who wants the bell?

AUDIENCE: I'll take it, I guess.

JESS RAY: All right. Can you hold that for a second? So what this bell is going to do is help us get rid of some of those negative ideas. Can you give it a ring? Oh yes. So that painful ringing you're hearing in your ears right now is actually just clearing up the air for us, making it so I can read your minds. Thank you. Stop that.

All right. Next we have this magic tone here. Who would like to give this a spin? Can you shake that around a couple of times? Spin it. Spin it with your wrist there, like-- you can go like this. There we go. All right. Perfect.

All right. It's feeling a little clearer here. I can start-- you can start getting things off your mind. Don't worry, I won't tell anybody what you're thinking. Oh, let's see what else. Let me channel the spirits. Help me out here. All right, I'm feeling good. All right.

So what we're going to be doing is, as I said, reading your mind. I'm going to be doing this by giving you cards, and I'm going to tell you what each of you are holding for the card. So I have some cards here. Well, I guess these are a little small. Let's see. Go a little bigger. Meh. Here we go. Let's-- this looks better.

All right. These are kind of heavy. Get rid of these junk ones up here, all the junk. All right. So I need your help for this. So what I want you to do is take the cards and cut the deck as many times as you want. So, basically, just going like that however much. Just don't actually shuffle them randomly.

AUDIENCE: All right. Here you go.

JESS RAY: All right, cool. So now I'm going to hand each of you a card. Don't let me see it. Feel free to look at it. There you go. All right.

So the reason I'm wearing this awesome onesie is this helps me sweat out the bad energy. I'm literally sweating right now. But there's one more piece that we need for this mind reading trick. The magic hat. All right.

See if this fits on my head. There we go. Where's the switch? All right. Turn it on. All right, I'm feeling good here. All right, you guys ready? All right.

So I do need a little help getting this trick started. So if you are holding a red card, can you just raise your hand? So no? Who's got the red card? Red, red. You don't have red? OK. All right. So the first one and the third one. All right.

So let me handle the mind reading abilities here. Now what I'm going to do is I'm going to go left to right and tell you what you're holding. Obviously, I know the color, but I'll tell you what suit it is, and also I will tell you what the number is.

So first card, obviously I know you have a red. Hmm. I'm feeling a diamond and also a four?

AUDIENCE: That was it.

JESS RAY: Yes. All right. All right. Good start, good start. All right. Got to-- got to think about what the next one is here. All right. So I know you had a black card. Let's see. Black of spades. Is it the ace of spades? Oh yes. There we go.

All right. So back to red. All right. This one, let's see. Red, diamond, two. All right, all right. We're doing good so far. Can I get the last two?

All right, let's see what we can do here. All right, black, club, four. All right. Last one, last one.

All right. Oh, it's going to be a tough one. Black, spade, eight.

[STUDENTS APPLAUD]

And if we had time, I could you mystify you and go through the rest of the deck, but we won't do that. So thank you guys very much. I hope your minds were blown. Yes. So me collect the cards back from you. Thank you. All right. Thank you. Now I can get out of this and stop sweating.

[STUDENTS APPLAUD]

JULIAN SHUN: It's pretty cool, right? So why does this actually work? To know why this trick actually works, we need to first study what a de Bruijn sequence is. So a de Bruijn sequence s of length 2 to the k is a cyclic bit sequence such that each of the 2 to the k possible bit strings of length k occurs exactly once as a substring in s.

So this a pretty long definition, so let's look at an example. So here is a de Bruijn sequence for k equals 3. So the length of this sequence is 8 because 2 to the 3 is 8. And you can see that each of the possible three-bit substrings occurs exactly once in this cyclic bit string of length 8. So it wraps around and you can consider this as a cyclic string.

So we see that 000 appears at position 0. 001 is at position 1. Then 010 is at position 6. 011 is at position 2. 100 is at position 7. 101 is at 5. 110 is at 4. And then 111 is at 3. So all of the 8 possible substrings of length 3 occur exactly once in this de Bruijn sequence.

So now we're going to create this convert table of length 8. In general, this will be 2 to the k. And here, k is 3. And in this convert table, what we're storing in each position is the index in the de Bruijn sequence where the bit string corresponding to that position starts in the de Bruijn sequence.

So here we see that convert of 2 is 6 because the bit string corresponding to 2 is 010, and that begins at position 6 in the de Bruijn sequence. We also see that convert of 4 is 7 because 4 is 100, and that begins at position 7 in the de Bruijn sequence.

Now we have this convert table. And recall that we're trying to compute the log base 2 of a power of 2. So hopefully you guys remember that. So the way to do this is we're going to multiply the de Bruijn sequence constant by this power of 2.

So let's say we're working with the integer 16, which is 2 to the 4. So we're going to multiply this de Bruijn sequence by 2 to the 4. And when we multiply by a power of 2, that's the same as left shifting. So that's going to left shift the de Bruijn sequence four positions to the left. And then now we want to see which of the eight possible substrings appears at the beginning of this sequence. And after we do the left shift, 110 appears at the beginning of the sequence.

And we want to extract this out, and we can do that by right shifting five positions. And 110 is just 6. And we can figure out where 5 starts in this de Bruijn sequence by looking it up in the convert table. We see that convert of 6 is 4. So the string 110 appears starting at position 4 in the de Bruijn sequence, and that means that we did a left shift by 4 in the first step, and that gives us the log base 2 of the power of 2, because the only reason why we did a left shift by 4 is because the power of 2 was 2 to the 4.

So this returns us the log base 2 of the integer that we started with. And one thing to note is that it's important to start with all 0's in this sequence here, because we're representing this as a cyclic bit sequence. So when we do a left shift, we need to make sure that the values that fill in on the right side are correct. So notice that in the sixth and seventh positions, we need 0's at the end when we overflow. So because the de Bruijn sequence starts with all 0's, when we do the left shift, it's automatically filling with 0's, giving us the correct substring.

So the magic trick that Jess did had 32 cards, and in that case k was equal to 5. And the cards were arranged according to a de Bruijn sequence of length 32. And each of the cards corresponded to one particular bit string of length 5. And the color of the card corresponded to the bit.

So when she asked you what the color of your card was, she could determine the bits corresponding to the first card in the sequence because she has the 5 bits corresponding to that card. And then with that she has some clever way to determine the rest of the cards. So that's how the de Bruijn sequence is related to the magic trick that you just saw. Any questions? Yes.

AUDIENCE: The de Bruijn sequence, do you need to do cyclic translation?

JULIAN SHUN: So there could be multiple de Bruijn sequences. We just need one particular de Bruijn sequence to make this bit trick work. Yes. So this example is just for k equals 3. And the code I showed you before, that was for k equals 8, so you can do up to 64-bit words. Yes.

AUDIENCE: How do we know that the sequence exists?

JULIAN SHUN: So there is a mathematical proof that says that. I can give you some pointers so that you can look at it after class. But there's a proof that says that for any length there is a de Bruijn sequence. Yes.

AUDIENCE: Sorry, I missed the procedure. So how exactly do you determine the log base 2?

JULIAN SHUN: So we have-- we're starting with some integer that is a power of 2. So when we multiply by that power of 2, it's left-shifting by the log base 2 of that. And then we can determine how much we left-shifted because we know-- we can just look at the first three bits of this sequence after we did the left shift, and then look at where that three-bit sequence appears in the original de Bruijn sequence before we shifted it.

And to do that, you can look it up in the convert table. This is what we did when we looked up the bit string 110 in the convert table. And that tells us that it starts in the fourth position. That means that we left-shifted by 4, and that means that the value of n was 2 to the 4. Does that make sense? Yes.

AUDIENCE: So just to clarify this only works if you multiply the sequence by a power of 2, then it gives you back which power of 2 it was?

JULIAN SHUN: Yes. So this only works if you're starting with a power of 2. So if it's not a power of 2, this doesn't work. Any other questions? Yes. So if it's not a power of 2, you can round it up to the nearest power of 2 using another bit trick that we saw earlier. And then you can use this bit trick here.

The performance of this bit trick is limited by the performance of multiplication and table lookup. So you have to do a multiplication by some constant, and then you have to do table lookup in this convert table. So a table lookup does a memory reference, which could be expensive. And nowadays there's actually a hardware instruction to compute this, so you don't actually have to implement this trick. But this trick is still pretty cool. And in the past this is how you would do it before there was a hardware instruction that came out.

So let's look at another problem. So this is the n queens problem. How many of you have seen this before? Yes. So many of you have seen this before. As a reminder, we're trying to place n queens on an n by n chessboard so that no queen attacks another queen. In other words, there are no two queens in any row, any column, or any diagonal.

And, commonly, we want to count the number of possible solutions to the n queens problem for a particular value of n. And in this example here, this is a valid configuration. You can check, for each of the queens, they can't attack any other queen on the board.

So one common strategy for implementing the n queens algorithm is to use backtracking. We're going to try placing queens row by row. We know that there can only be one queen per row, so we just need to determine which position in that row the queen will appear in. And then if we can't place a queen in any row, then we backtrack.

So, for example, in the first row, we'll just place the queen in the first position, because there's no queens on the board yet, so the first position is valid. For the second row, we're going to try to place in the first position, but we can't place it there because then it will attack the first queen. And then the second position is also invalid, so the third position is where we place the second queen.

Now, for the third row we're going to check the positions until we get to one that's valid, and this is going to be the fifth position. Do this again. Here we can do it in the second position. For the fifth row, let's see where this is going to end up. OK. So it goes in the fourth position. What about the sixth row?

Whoops. So all of the eight positions are invalid, because if we place the queen in any of those positions, it's going to attack one of the queens that we already placed. So now we're going to backtrack. We're going to find another position for the fifth queen. So let's try some more positions. So we can place it at the end. Now we try again.

All right. So, unfortunately, we couldn't find a position for the sixth row again. We have to backtrack. But we already tried all the positions in the fifth row, so we backtrack to the fourth row. And you get the idea. And then whenever we find a configuration where all eight queens are valid, then we increment some counter by 1. And at the end we just return this counter, which tells us the number of solutions to the n queens puzzle.

So you can implement this quite easily using a recursive procedure. You can implement this backtracking search. But one question is how should we represent the board to facilitate efficient queen placement? So one way to represent the board is to use an array of n squared bytes. And for each byte, we just have a 1 if there is a queen in that position, and 0 otherwise. Is there a better way to represent the board?

AUDIENCE: You can track all of the bits such that a 1 bit represents a queen at some place on the board?

JULIAN SHUN: Yes. So that's a good answer. So instead of using bytes, we can use bits, because the value can only be 0 or 1. We only need one bit to represent that. So we can just have an array of n squared bits. Is there a better way to do this? Yes.

AUDIENCE: You could just say in each row where a queen is with a byte?

JULIAN SHUN: Yes. So good answer. So a better way to do this is to just use an array of n bytes. Because we know that on each row there can only be one queen, so we just need to store the position of that queen. So we have an array of n bytes, one byte for each row, and then you just used the byte to store the position of the queen in that row.

It turns out, to implement this algorithm, there's a even more compact representation, which is to use three-bit vectors of size n, 2n minus 1, and 2n minus 1. So let's see how this works. So the first bit vector we're going to use is of length n. We're going to call this the down vector. And the down vector just stores a 1 in the columns that have a queen in it and 0 in the columns that are empty.

And then when we want to check whether placing a queen is safe in any position, we first have to check whether that column is empty. And you can do this by ANDing the down bit vector with 1 left-shifted by c, where c is a column where you want to place the queen. And if that's nonzero, that means there's already a queen in that column and you can't place it.

Otherwise, we're going to have to do another check, and we're going to create this other bit vector called left. The length of this bit vector is 2n minus 1. And it stores a 1 in the diagonal that has a queen in it, and 0's otherwise. And there are 2n minus 2 possible diagonals.

And then now, when we want to place a queen in row r and column c, we can check whether it's safe by doing left ANDed with 1 left-shifted by r plus c. And this is going to be nonzero if there is already a queen in that particular diagonal. So in that case, we can't place a queen there.

And, otherwise, we're going to do a final check using this right bit vector, which is essentially the same but we're looking at the diagonals going down to the right. So, again, we have a 1 in the diagonals that have a queen and 0's otherwise. And then now the check is going to be right ANDed with 1 left-shifted by n minus 1 minus r plus c. And if a particular candidate passes all three of these checks, then we know that there's not going to be a conflict and we can place the queen in that particular position.

So this is a bit vector representation. You actually still have to write the code to count the number of queens using this bit vector representation, and it's actually an interesting exercise. So I encourage you to try to do this at home. But I just told you about the bit vector representation. So any questions? Yes.

AUDIENCE: Could you just repeat what the down vector bit hack was for figuring out [INAUDIBLE]?

JULIAN SHUN: Yes. So the down vector, it stores a 1 in the columns that have a queen in it and 0's otherwise. And what you do is, if you want to place a queen in column c, you first create the mask 1 left-shifted by c. And then you AND it with a down vector. And that's going to be nonzero if there's a queen in that column. Any other questions? Yes.

AUDIENCE: Why isn't there a horizontal one?

JULIAN SHUN: So it turns out that you don't need. Just these three checks is enough to guarantee--guarantee that you can place a queen in a position if it passes all three of the checks. Yes. So a fourth check would just be redundant.

AUDIENCE: So we don't need a horizontal one because we're not placing two queens in the same row.

JULIAN SHUN: Yes. That's true. Good point. Yes. So we're only placing one queen in each particular row.

So let's look at another problem. This is called population count, or pop count for short. And the problem here is we want to count the number of 1 bits in some word x. Here's a way to do this that repeatedly eliminates the least significant 1 bit in a word.

So we have this for loop where r is initialized to 0. And we're going to repeat this loop until x becomes 0. And then each time we go through this loop, we increment r. And inside the loop we're going to set x equal to x ANDed with x minus 1. And this is going to clear the least significant 1 bit in x.

So let's look at an example. So let's say we have this value here for x. Well, to get x minus 1, we flip the rightmost 1 bit in x from a 1 to 0. And then we fill in all of the bits to the right of that with 1's. And then now when we AND those two things together, we're going to copy all of the bits up to the rightmost 1. And then for the rightmost 1, we're going to zero it out because we're ending with a 0. And then all of the bits to the right of that are still going to be 0. So x ANDed with x minus 1 is just going to get rid of the least significant 1 bit.

And then we repeat this process until x becomes 0. In that case we've already eliminated all the 1's and we know the answer, which is stored in r. Questions? So this code will be pretty fast if the number of 1 bits is small, but the running time is proportional to the number of 1 bits in a word. So in the worst case, if most of the bits are set to 1, then you're going to need a lot of iterations to run this code.

So let's look at a more efficient way to do this. This is to use table lookup. So we're going to create a table of size 256, which stores for each 8-bit word the number of 1's in that 8-bit word. So we have all possible 8-bit words stored in this table.

And then now, to get the number of 1 bits in x, for every 8-bit substring in x, we're going to look it up in this count table and add it to r. And then we're going to right-shift x by 8 so that we can get the next word. And then when x becomes 0, we know we're done. So that's table lookup. And the performance here depends on the size of x. If we have a 64-bit word, we need to do this at most eight times, whereas in the initial code we might have to do it 64 times if we had 64 1 bits.

The cost of this code is bottlenecked by the memory operations, because this table here is stored in memory. So every time you access it you have to go to memory to fetch the value there. And here are some approximate costs for accessing memory in various levels of the hierarchy. If something's stored in register, it's very fast. It only takes you 1 cycle. If it's stored in L1 cache, it's about 4 cycles, L2 cache about 10 cycles, L3 cache about 50 cycles. And then, finally, if you have to go to DRAM because it's not in cache, it's much more expensive, 150 cycles. It's an order of magnitude slower than doing something-- fetching something that's already stored in a register.

So let's now look at a third way to do population count where we don't actually have to go to cache or DRAM. Essentially, we can do everything in registers. So here's how you do it. So we're going to create these five masks-- or six masks, from M0 up to M5. And these masks-- the values of these masks are shown in the comments here. In this notation here, x to the k just means x repeated k times. So the mask M5 has 32 0's, followed by 32 1's. The mask M0 has the bit string 01 repeated 32 times, and so on.

After we create these masks, we're going to execute these six instructions at the bottom, and this is going to give us the number of 1's in the word. So let's do an example to see how this works. So let's say we start with this bit string here. In the first step, what we're going to do is we're going to AND x with the mask M0.

And then we're also going to AND x right-shifted by 1 with the mask M0. and recall that the mask M0 is just 01 repeated 32 times, and therefore the mask is essentially extracting all of the even bits. So x ANDed with M0 gives us all of the even bits. And then when we right-shift x by 1 and AND it with M0, that's going to give us all the odd bits.

And then we're going to line those two things up and add them together. And the result of doing this is it's going to tell us for every group of two bits the number of 1 bits in that group. So now for each of these pairs of bits, it's telling us how many of them are 1. So in the leftmost group here, we add two 1's. So the result of adding 1 and 1 is 1 0, which is 2. For the rightmost group, we have two 0's, and the count there is 00. And this is the same for all of the other groups.

So this gives us the number of 1's in every pair of positions. Now we're going to AND the result with M1. And we're going to right-shift it by 2 and also AND it with M1 and add those two things together. And M1 is a mask that will give us the bottom two bits in every group of four bits. So when we right-shift x by 2, that's giving us the top two bits.

And then now we add those together, and it will give us the count of the number of 1 bits in every group of size 4. And these counts are stored in the result here now. So you can verify that each of these groups has the count of the number of 1 bits. So, for example, we have 100 here. And this is correct since there are four 1 bits.

Now we do this again with the mask M2. That's going to give us the counts for all groups of size 8. Then we go to groups of size 16. And then, finally, we add these two together, giving us the number of bits in this group of size 32. And this is actually the pop count. So the value here is 17. And you can verify that there are indeed 17 1's in the input word x. Any questions?

So the performance of this code, which is based on parallel divide and conquer, is going to be proportional to log base 2 of w, where w is the word length. Because on every step I'm doubling the size of my groups. And after I do this log base 2 w times, I have the whole group.

In the first two instructions that I executed here, I have to actually do the AND separately for x right-shifted by 1 and x, and also x right-shifted by 2 and x, and then add them together, because there is an overflow issue. The overflow issue is that the size of the groups here might not be large enough to actually store the count of the number of 1 bits in that group. But once I get to the larger groups, the count can always be stored in a group of that size and I don't need to worry about overflow. So for the last four lines, I can actually save one instruction. I don't need to do the AND twice.

So it turns out that most modern machines nowadays have an intrinsic pop count instruction implemented in hardware, which is faster than anything you can code yourself. And you can access this pop count instruction via compiler intrinsics, for example in GCC or Clang. And in GCC, it's __builtin_popcount. One warning though is that if you write this code using these intrinsics, if you try to compile the code on a machine that doesn't support it, your code isn't going to compile. So it makes your code less portable. But this intrinsic is faster than the parallel divide and conquer version.

So one question is, how can you get the log base 2 of a power of 2 quickly using a pop count instruction? So instead of using the de Bruijn sequence trick. Yes.

AUDIENCE: You decrement then you pop count.

JULIAN SHUN: Yes. So what you do is you subtract 1 from the power of 2, and that's going to flood all of the lower bits with 1's. And then now when you execute pop count, it's going to count the number of 1's, and that gives us the log base 2 of the power of 2. So good answer.

So those all the bit tricks I'm going to be talking about today. There's a lot of resources online if you're interested in learning more. There's this really good website maintained by Sean Eron Anderson. There's also the Knuth's textbook, which has some bit tricks in there. There's a chess programming website which has a lot of cool bit tricks. Some of those are used in implementing chess programs. And then, finally, this book called *Hacker's Delight.* So we'll be playing around with many of these bit tricks in project 1, so happy bit hacking.