Episode 7.06 – Stupid Binary Tricks

Welcome to the Geek Author series on Computer Organization and Design Fundamentals. I'm David Tarnoff, and in this series, we are working our way through the topics of Computer Organization, Computer Architecture, Digital Design, and Embedded System Design. If you're interested in the inner workings of a computer, then you're in the right place. The only background you'll need for this series is an understanding of integer math, and if possible, a little experience with a programming language such as Java. And one more thing. This episode has direct consequences for our code. You can find coding examples on the episode worksheet, a link to which can be found on the transcript page at intermation.com.

We are dedicating this episode to some of the little tricks we can do with bitwise operations. Many of the exercises are not really of much use – they're more like exercises to flex your digital representation muscles. It is modeled after a homework one of my digital logic instructors assigned to me many years ago. It relies heavily on an understanding of 2's complement representation, so if you need a refresher, you might want to go back and listen to Episode 3.03 – An Introduction to Two's Complement Representation. Go ahead, we'll wait. Welcome back!

My old assignment was to create a library of functions that took one or more integer parameters and returned a single integer result. Inside the function, you were limited to using only integer constants 0 through 255 (0xFF) and the bitwise operations ~ (the bitwise inverse), & (the bitwise-AND), | (the bitwise-OR), ^ (the bitwise-XOR), + (addition), << (the left shift), and >> (the arithmetic right shift). Some of the problems restricted the set of allowed operators even further. Your grade was based on how many operations it took you to solve each of the problems.

We were forbidden from using any programming structures such as if-statements or loops. Okay, forbidden is a rather harsh word, but we lost points if we used one. Each function had to be self-contained and not call any other functions. We assumed that the processor on which the functions were to be run represented integers using 32-bit two's complement. As a side note, not all architectures use 32-bit to represent integers. Many compilers have a mechanism for us to determine how many bits are used to represent different variables. C, for instance, uses a function called sizeof() that can be used to compute the number of bits in a data element. Our assignment further disallowed the use of arrays, classes, or objects. We also were not allowed to use type casting to change the binary representation. You might say that an assignment like this is a waste of effort since we have programming structures like if-statements and loops. This is not entirely true. One thing that the bitwise operations do for us is create consistency in execution time. The number of iterations of a while loop changes based on the value passed to the operation. The bitwise operations always execute the same number of instructions, a number which more than likely will be far less than the while loop. This aids certain computing solutions that require consistent and dependable execution times, something that a while loop implementation cannot offer.

If-statements and while loops also require the execution of a low-level instruction called a branch. In many cases, branches affect the operations of a processor component referred to as a pipeline. Removing branches from our code has the potential to provide incremental improvements in performance.

Let's begin with a simple problem: find the least significant one in an integer. This is the same as finding the largest power of two that is a divisor of the integer. It turns out that if you know how two's complement representation works in the computer, this is a simple task. Remember the shortcut from Episode 3.03 to take the negative of an integer? It went something like this. Starting from the right most bit, copy consecutive zero bit values until you reach your first binary one. Copy that 1 too. If the least significant bit is a one, then only copy that bit. Then invert all the remaining bits to the left.

That means that a non-zero integer and its corresponding negative representation share exactly one position where both contain a one. To the right of that position, both values have zeros. To the left of that bit position, the bits are inverses of each other. By using the bitwise-AND, we can strip out all the ones except the least significant one. This gives us the expression:

```
LeastSignificantOne = -x & x
```

Note that this works the same whether the integer is positive or negative. A zero will return a zero. Finding the most significant one in an integer is a little more challenging. Remember that in two's complement, the most significant bit is the sign bit, so let's stick with positive integers here. How are we going to determine the most significant one in a binary integer? Well, there's a trick, but it's not graceful. It involves copying the most significant one to every bit position to the right of it. We do this with a series of arithmetic right shifts and bitwise-ORs.

Begin by creating a duplicate one in the bit position immediately to the right of each one in the original value. Assuming the variable we are working with is named 'x', the code to create a duplicate one to the bit position immediately to the right of each original one in the integer is:

```
x = x | (x >> 1)
```

Now that each one has been paired with a one to the right, we can duplicate the pairs of ones to the positions to the right so that each original one has three ones to its right. The code to do this is:

```
x |= x >> 2
```

Now we are going to use the same process to duplicate groups of four ones to the right of each bit position with a one in it so that original one has seven ones to its right.

```
x |= x >> 4
```

Next, duplicate groups of eight ones to the positions to the right so that each original one now has fifteen ones to its right. The code for this is:

```
x |= x >> 8
```

Since we are working with 32-bit values, we need to do this one last time so that our stream of ones to the right of each original one is at least 31 bits long. This is done by duplicating groups of 16 ones to positions to the right. Note that most of the ones being shifted right end up being truncated. The line of code is:

```
x |= x >> 16
```

The right shifts and bitwise-ORs basically over-wrote all of the bit positions to the right of the most significant one with ones, so the number we are left with has all zeros to the left of the most significant one and all ones to the right.

Adding one to the modified value of x turns all the ones to zero and places a carry in the position immediately to the left of the original leftmost one. Therefore, all we need to do is increment x by one.

This will take all of the ones, turn them to zeros, and put another carry in the bit position immediately to the left of the most significant one (x++).

If we perform a logical shift right by one position on this value, we have our most significant one. The code for this is:

```
x >>>= 1
```

If we don't have a logical shift right available to us and can use only an arithmetic shift right, then we need to perform a bitwise-AND to clear the most significant bit. This is for the case where the leftmost one of our binary value is the bit position immediately to the right of the sign bit. The logical shift right would give us a result of two ones followed by thirty zeros after we perform the arithmetic shift.
Because the previous challenge required a positive number, maybe our next challenge should be to determine the absolute value of an integer using bitwise operations. Determining the absolute value of an integer using bitwise operations is much quicker using only two lines of code. In fact, it's quicker than trying to do it with an if-statement and has more consistent performance when run on a processor.
Start by creating a variable "sign", where all thirty-two bits equal the sign bit of x. Another way of saying this is that the variable "sign" equals 0 if x is positive and -1 (a binary value of all ones) if x is negative. To make this variable, simply perform an arithmetic shift right by thirty-one positions. This will copy the sign bit across the whole 32-bit integer. The code to do this is:

```
sign = x >> 31
```

If x is non-negative, we want to return x. If x is negative, however, we want to convert it using the two's complement method of flipping all the bits and adding one, which by the way, is the same as subtracting one, then flipping all of the bits. The subtracting one is easy – we just add the sign variable which is zero for positive numbers and negative one for negative numbers.
But what about flipping all the bits? Remember that the XOR is a bit like a programmable inverter. If we do a bitwise-XOR using sign as our mask, when x is negative, the mask is all ones and the bitwise-XOR will invert every bit. When x is positive and the mask is zero, all the bits of x are left unchanged. That gives us the code:

```
absoluteValueOfX = (x + sign) ^ sign
```

There is a case where this bitwise operation does not work, and it's due to the mechanics of two's complement representation. In two's complement, the most negative value is represented in binary as a one followed by all zeros. For thirty-two bits, this is negative two to the 31st power or -2,147,483,648. The most positive two's complement value in thirty-two bits is one less than two to the 31st power or 2,147,483,647. That means that the most negative value in two's complement does not have a corresponding positive value. In that case, our absolute value code will return the original negative value.

Now that we have the code for determining both the least significant and most significant ones and the ability to take the absolute value of a number, we can quickly determine if an integer is a power of two. We do this by exploiting the fact that a positive power of two in binary has a single bit set.
We start by taking the absolute value of the integer. Next, create a variable to contain the least significant one of the absolute value and another variable to contain the most significant one. If we take the bitwise-AND of the most significant one with the least significant one and the result is not equal to zero, we know that the same bit is both the least significant one and most significant one and the original integer was a power of two.

It turns out that there is a much quicker way to do this, but it introduces a problem. Subtracting one from a power of two turns the single one in the power of two position into a zero and all the zeros to the right of that one turn into ones. That means that no bit position has a one in both the power of two value and the value that is one less than that power of two. If we take the bitwise-AND of the value with the value minus one, a power of two will give us a result of zero. The code to figure this out is simply
`x & (x - 1)`

There are two items of note here. First, the value returned by this expression will be zero when x is a power of two and some non-zero value if x is not a power of two. This could be confusing. Second, zero will be identified as a power of two with this code, which is not correct. Zero is not a power of anything, except zero.

Oh, and if it bothers you that we used subtraction, which based on our earlier premise, is not an allowed operation, simply replace x – 1 with x + ~0. Adding the bitwise-inverse of 0 is the same as subtracting one.

I know this episode has gotten a bit tedious, but let's do one last example before we go. Let's determine if a two's complement integer, x, can fit inside of n bits. For example, the integer 15 along with its sign bit will fit into five bits, but will not fit into four bits.

The operation goes something like this. Begin by using an arithmetic shift right on x by n minus one bits to copy the sign bit to the entire number. The code to assign this value to the variable "sign" is:
`sign = x >> (n - 1)`

If x requires more than n bits to be represented, then a shift of n minus 1 positions right won't get all of the magnitude bits shifted out. How do we test for this? Well, we simply need to check if the result is zero for a positive value or negative one for a negative value. For both zero and negative one, all bits are the same. If all bits of a value are the same, then an arithmetic shift right by one position should also result in the same value. An exclusive or will test for this. The code becomes:
`sign ^ (sign >> 1)`

If the result is zero, x will fit in n bits. Non-zero values indicate that x will not fit.
Well, that was quite a long episode. If you found any of it confusing, please visit us at intermation.com where you will find the episode transcript and worksheet along with links to our Instagram, Twitter, Facebook, and Pinterest pages. Until the next episode, remember that while the scope of what makes a computer is immense, it's all just ones and zeros.