Episode 7.05 – Flipping Bits using the Bitwise Inverse and Bitwise-XOR

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.

Up to this point, we've been manipulating bits by clearing them and by setting them. There is one more way we can bash bits – we can invert them. There are two ways to invert the bits of an integer. First, we can flip all the bits. This is done using the bitwise inverse. In our code, we invert the bits of an integer by placing the tilde operator in front of it. This can be done in front of a variable name, a defined constant, or even a number. For example, ~0 (read tilde zero) will return an unsigned integer with all of the bits set to one. If we do this and output the variable in unsigned hexadecimal or binary, we have a quick way to determine how many bits your computer represents an integer with.

As an aside, JavaScript does not allow us to define a variable as an unsigned integer. Therefore, ~0 will display as negative one in our browser window, not 0xffffffff like we want. We can create an inelegant work-around called a "kludge" though. Remember from our twos complement discussion that the most significant bit is considered a sign bit. When this bit is set to one, JavaScript will output the negative value. The kludge is to make the most significant bit a zero so that the variable is no longer negative and it will display the remaining bits as ones. We can use the logical right shift, the operator that is three greater than symbols next to each other, to shift the bits right by one position and fill in from the left with a single zero. Executing the line of JavaScript **`document.write(((~0)>>>1).toString(2))`** will output thirty-one ones in a row. As a side note, the toString() function will convert an integer to the base we ask for by putting that base in parenthesis after "toString".

The bitwise-XOR, represented in our code using the caret or wedge as the operator, uses a mask to invert specific bits of an integer while leaving the other bits alone. Recall the truth table for a two-input XOR. If the bits are the same as is the case for the top and bottom rows of the truth table where A equals zero and B equals zero and where A equals one and B equals one, the output is a zero. In the middle two rows of the truth table where the inputs are different, A equals zero and B equals one and A equals one and B equals zero, the output is one.

If we cover up the bottom two rows of this truth table leaving visible only the rows where A equals zero, we see that the output is equal to B. If we cover up the top two rows leaving only the rows where A equals one visible, the output follows the inverse of B. It almost seems like the two-input exclusive-OR behaves like a programmable inverter. If A equals zero, B is passed through to the output untouched. If A equals one, B is inverted at the output.

This concept transitions nicely to the use of a mask with the bitwise-XOR. If we perform a bitwise-XOR, the bit positions in the mask with zeros will pass the original value through and the bit positions in the mask with ones will invert the original value. For example, lets say we have a byte equal to the binary

value 10101010 and a byte-wide mask equal to 00001111. If we take the bitwise-XOR of these two values, the leftmost nibble of the value, 1010, is left untouched while the least significant nibble is inverted to 0101. This gives us the result 10100101.

Let's do this again with an example. There are a number of formats for representing a color using binary. A few of these formats use four eight-bit channels: one for red, one for green, one for blue, and a fourth channel called alpha. The alpha channel represents the degree of transparency for the color. When the byte representing the alpha channel is equal to all ones or 0xFF in hexadecimal, the color is rendered fully opaque. When the alpha channel is equal to zero, the color is transparent regardless of the value to which the red, green, and blue channels are set. Values of alpha between zero and hexadecimal 0xFF present incremental levels of transparency (or opacity) between transparent and opaque.
One of the 32-bit formats arranges these four bytes in the order alpha, red, green, and lastly, blue. This is often referred to as ARGB. As an example, the hexadecimal ARGB value 0x80FFFFFF is white (full red, full green, and full blue), with a transparency of around fifty percent.

Let's say we want to create a blinking effect where every second we flip the bits of the alpha channel back and forth between all ones and all zeros. Leaving aside the level of annoyance a feature like this would create, we can do it with the bitwise-XOR. All we would need to do is detect when the second timer was finished, perform a bitwise-XOR with the hexadecimal mask 0xff000000, then start the second timer again.

Let's look at another example. Assume a byte is used to control the warning and indicator lights on an automotive dashboard. The following is a list of the bit positions and the indicators they control.

bit 7 – Oil pressure indicator
bit 6 – Temperature indicator
bit 5 – Door ajar indicator
bit 4 – Check engine indicator
bit 3 – Left turn signal
bit 2 – Right turn signal
bit 1 – Low fuel indicator
bit 0 – High-beams indicator

If the driver put the emergency flashers on, we should do a bitwise-XOR with a bit mask that causes the left and right turn indicators to invert their values every time a timer times out. Why would we want to use a bitwise operation? Well, it's vital to leave the other indicators, such as the oil pressure and temperature, alone.

Remember that the bitwise XOR uses a mask with ones in the positions to be toggled and zeros in the positions to be left alone. To flip bits 3 and 2 between on and off, the mask should have ones only in those positions. Therefore, the mask to be used with the bitwise-XOR is 00001100.

These two examples may seem a bit academic, but bitwise-XORs have many important applications including their use in detecting errors. For example, Ethernet uses the cyclic redundancy check or CRC to detect if a message has been corrupted. We're only going to dig deep enough into the details of creating a CRC so that we can describe how the bitwise-XOR is used.

A CRC is a bit like the modulo or remainder function. When we take the modulo of a number, it returns the remainder. We encounter a problem when we try to perform the modulo function on an incoming message like an Ethernet frame. To perform a long division, the full message must be received and stored in a high speed memory location called a register. This is okay if we're performing the modulo on a 32-bit or 64-bit value, but Ethernet frames can over 12,000 bits long. We're not going to have a register that we can perform math on that large an integer.

The problem is that long division depends on subtractions and subtractions depend on borrows. It's the borrows that are causing the problem. If we could perform a borrow-less subtraction in order to generate a pseudo-modulo result, we could perform this operation as the message is being received. In a borrow-less subtraction, zero minus zero is zero, zero minus one is one, one minus zero is one, and one minus one is zero. Take a look at that last sentence. You'll see that when the digits are the same, in other words, zero minus zero and one minus one, the result is zero. When the digits are different, in other words, zero minus one and one minus zero, the result is one. That's exactly how the two-input exclusive-OR works! That means that a CRC can be computed much the way a long division is computed by replacing the subtractions with bitwise exclusive-ORs. Without the borrows, we don't have to wait for the entire message to be received before we can start computing the CRC. We can perform the operation as the message is being received. The details are beyond the scope of this episode, but when we begin discussing error detection in serial protocols, you can believe we will take a deeper dive.

The next episode is going to drive home the use of bitwise operations by performing a number of cool binary tricks. For episode transcripts, worksheets, links, or other podcast notes, please visit us at intermation.com where you will also find 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.