Episode 7.01 – The Need for Bitwise Operations

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.

We are going to take a couple of episodes to cover this next topic, not because it's difficult, but rather because of its importance to computing. The ability to manipulate numbers at the bit level using software is vital to many of our computational algorithms. If you've been following this series, you've seen a little bit of this already. In Episode 2.02 – Unsigned Binary Conversion and again in Episode 3.04 – The Application of Two's Complement, we presented the bit shift operators and showed how they quickly accomplish multiplication and division by powers of two by shifting the bits in our binary numbers left and right.

If you're familiar with programming, you know that most data manipulation is done in software using mathematical operations such as multiplication or addition. Some applications, however, require the examination or manipulation of data at the bit level. For example, most programming languages have the ability to generate a random number. In Java, we can generate a random number using the nextInt() function provided to us by the Random class. Suppose, however, that we want to generate a random odd integer. Most of us probably learned to do this by first generating a random integer, next, multiplying that integer by two to force it to be even, and lastly, adding one to force it to be odd. We could do this using bit shifts. If you recall, a bit shift one position to the left is equivalent to multiplying by two. Substituting this bit shift for the multiplication step in our process doesn't really change much. But what does an odd number look like in binary? Well, the only bit position that represents an odd value within a binary integer is the least significant bit. If the least significant bit is one, we have an odd number. If the least significant bit is zero, the number is even. So all we really need to do to guarantee our random integer is odd is set the rightmost bit to a one. Conversely, we could generate a random number and force it to be even by clearing the least significant bit. An understanding of bit manipulation makes this task quick and efficient.

This means that in addition to modifying our binary values using bit shifts, we can also manipulate data at the bit level by changing the settings of the individual bits. To do this, we use a set of functions called bitwise operations, and the typical programming language provides operators to support them. The term bitwise operation refers to the setting, clearing, or inversion of individual bits within a binary number. This is done by executing one of the logical operations, AND, OR, NOT, or exclusive-OR, on pairs of bits that share the same position within two binary numbers. The bits are paired up by matching their bit position, performing the logical operation, and then placing the operation's outcome in the corresponding bit position of the result.

While instructors often encourage questions, one of the questions typically at the bottom of our list of favorites is, "Do we really need to learn this stuff?" Answers such as, "Intellectual effort is good for the mind," or, "It makes you a well-rounded person," or, "It develops critical thinking," seem to fall short. In

the case of bit manipulation, however, the ability to examine and modify individual bits is critical to becoming an effective programmer.

If you're maintaining a network, some protocols such as IPv4 rely on bit manipulation to dissect the addressing. If you're designing a web page and need to define a 24-bit color, you'll have better control by using bit manipulation. If you're an embedded systems engineer, you'll need bit manipulation to control and configure your input and output devices. If you're programming high-performance applications, you can take advantage of the speed with which bitwise operations perform to make your applications faster. If you're creating a user interface, you can highlight text by flipping some of the bits used to represent the background color to get a new background color. To un-highlight the text, simply flip those same bits back to their original value. Compression and error checking algorithms often rely on the manipulation of the bits representing the data. Settings such as who gets to see a shared file are often stored together in a single integer referred to as a bitmap. A particular individual's privileges are represented by a bit in a specific location. Either writing or reading that information from the bitmap requires bit manipulation. Interpreting the data in blocks of information ranging from caller id to a GIF image requires bit manipulation. Need I go on?

Let's start with the color example. Open any application that allows the user to select a color. If that application uses 24-bits to represent a color, then there are eight bits that identify the level of red, eight bits that identify the level of green, and eight bits that identify the level of blue. Sometimes, eight more bits are used so that the user can also define the amount of transparency, referred to as the alpha channel. When editing these colors, applications will often display the values in decimal giving us a range from 0 to 255, the limits of an eight-bit value.

When storing these values, the application takes these three or four bytes and concatenates them into a single integer. Ignoring the alpha channel for now, the three bytes representing red, green, and blue, are joined into a single value where the rightmost eight bits, bit positions seven down to zero, represent blue, the middle group of bits, bit positions fifteen through eight, represent green, and the most significant eight bits, positions twenty-three down to sixteen, represent red. Because humans are more efficient at reading hexadecimal values, these colors are usually represented with a six-digit hexadecimal value. For example, the color light salmon is represented with the 24-bit hexadecimal value 0xFFA07A. This means that the pixel displaying light salmon does it with a level of 0xFF for red, a level of 0xA0 for green, and a level of 0x7A for blue.

Maybe the default color of white smoke, hexadecimal value 0xF5F5F5, which has equal levels of hexadecimal 0xF5 for red, green, and blue, is a little too drab for our client. They want to lower the blue a bit from 0xF5 to 0xDC to get beige, the hexadecimal value for which is 0xF5F5DC. That's okay, but it's still not vibrant enough for them. So we lower the green level to hexadecimal 0x4E and get the 24-bit color 0xF54EDC. Ugh! Purple pizzazz is too vibrant! Let's lower the red and see what we get. Changing the red level to a hexadecimal 0x60 gives us the 24-bit color 0x604EDC. Majorelle blue. Not great, but it's a start. Manipulating a 24-bit color at the bit level gives us the ability to adjust red, green, and blue levels independently of one another.

Bitwise operations allow us to manipulate the bits of a single integer such as this color value. In order to do this, we need to know two things: first, to what values do the bits need to be changed, and second, which bits are we changing. As for the first item, there are three types of bit-level operations: clearing bits to zero, setting bits to one, and inverting or toggling bits from one to zero or from zero to one. We clear bits using the bitwise-AND operation and set bits using the bitwise-OR. There are two operations

that allow us to invert the bits. One of them, the bitwise-inverse or NOT, will invert all of the bits of an integer, while the second, a bitwise-exclusive-OR, allows us to invert only selected bits within a binary value.

For three of these operations, the bitwise-AND, the bitwise-OR, and the bitwise-exclusive-OR, a bit mask is required. The term mask is important here. No, we're not implying that we are disguising the binary number like Batman or Darth Vader. Mask in this context is more like a silk-screen mask, where some areas are protected from being modified while other areas are modified. A bit mask is a binary value that has the same number of bits as the binary value we wish to alter. It has a pattern of ones and zeros that defines which bits of the original value are to be changed and which bits are to be left alone. In our next three episodes, we will discuss each of the three types of bitwise operations, clearing, setting, and toggling, show how their masks are to be defined, present examples where these operations are used, and discuss the code used to accomplish these operations.

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.