Episode 7.03 – Coding 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. 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.

Way back in Episode 2.2 – Unsigned Binary Conversion, we introduced three operators that allow us to manipulate integers at the bit level: the logical shift left (represented with two adjacent less-than operators, <<), the arithmetic shift right (represented with two adjacent greater-than operators, >>), and the logical shift right (represented with three adjacent greater-than operators, >>>). These special operators allow us to take all of the bits in a binary integer and move them left or right by a specified number of bit positions. The syntax of all three of these operators is to place the integer we wish to shift on the left side of the operator and the number of bits we wish to shift it by on the right side of the operator.

In that episode, we introduced these operators to show how bit shifts could take the place of multiplication or division by powers of two. The difference between the arithmetic and logical right shifts is what fills in the most significant bits on the left side as the bits move right. In the case of an arithmetic right shift, the most significant bit is duplicated. Remember that the most significant bit is a sign bit in two's-complement. By using an arithmetic shift right, positive numbers stay positive and negative numbers stay negative.

In the case of the logical shifts, the vacated bits are filled with zeros. This gives us the ability to move blocks of bits right without introducing new logic ones into the integer. This would be important when it comes to manipulating patterns of ones and zeros like those found in an RGBA color or a bitmap defining user privileges.

In Episode 7.02 – Clearing Bits using the Bitwise-AND, we discussed a number of examples of how performing a bitwise-AND of an integer with a binary mask allows us to clear individual bits of that integer while leaving the other bits alone. This discussion would be of only theoretical interest if we couldn't use these operations in our code. Fortunately, programming languages such as Java give us the opportunity to use all of the bitwise operations, AND, OR, exclusive-OR, and the bitwise inverse, in our code. At this point in our series, we've only demonstrated the use of the bitwise-AND, so this episode will focus on programming with that operation.

Typically, a single ampersand, the symbol found on the US QWERTY keyboard as a shifted 7, is used as the operator identifying the bitwise-AND. For example, value & 1 would perform a bitwise-AND between the variable "value" and the bitmask 1. This operation clears all bits except the least significant bit, the bit that identifies whether value is odd or even. The conditional statement if(value & 1) would evaluate to a "true" (a non-zero value) when the integer stored in the variable value is odd. The same

expression would evaluate to “false” (a zero value) if the variable value is even. You can find an example of this code written in JavaScript on the episode worksheet.

Now let’s combine the bitwise-AND with the bit shifts to do something useful. In Episode 7.01 – The Need for Bitwise Operations, we showed how computers use 24-bits to represent a color. The right-most or least significant eight bits represent the color’s blue level from 0 to 255. The group of eight bits immediately to the left of blue represents the amount of green from 0 to 255. The group of eight bits immediately to the left of green represents the amount of red from 0 to 255. We can use the right shift operators along with the bitwise-AND operator to separate these colors into three variables.
First, the simplest color to pull from the 24 bits is the level of blue. We do this by simply clearing the bits representing red and green. Remember that the bitmask used with the bitwise-AND places ones in the bit positions where we want to leave the original value of the bits and zeros in the positions we want to clear. If we want to clear all of the bits except for the least significant eight bits where the blue level is located, then we want a mask that has ones only in those bit positions. Remember that in languages based on the C syntax such as Java, the prefix 0x is used to represent a hexadecimal value, so our mask in hexadecimal would be 0xff. That gives us a simple equate statement blue = color & 0xff.
Green and red will be only a little more difficult to extract. Remember that like blue, the red and green levels are also represented with eight bits. They’re just not in the least significant bit positions…yet. If we perform a logical shift right by eight, then the blue bits will be shifted out and the green bits will be in their place. A bitwise-AND with the bitmask 0xff will clear the red bits and leave only green. This results in the equate statement green = (color >>> 8) & 0xff. Red is extracted similarly except that we need to shift right by an additional eight bits. The expression for this is red = (color >>> 16) & 0xff.
Episode 7.02 presented another example – the use of a subnet mask to identify the subnet of an IPv4 address by clearing the bits of the host id. The pattern of ones and zeros contained in an IPv4 subnet mask is configured such that the most significant bit positions, the ones representing the subnet, are set to one while the rightmost bits, where the host id bits are located, are zeros.

So what does our code look like? When represented in binary, both the IPv4 address and the subnet mask are 32-bit numbers. For our code, let’s assume that we have been given the subnet mask in a variable named subnetMask and the IPv4 address in a variable named ipAddress. The subnet id can be found with the code subnetMask & ipAddress. It’s that simple.

In later episodes, we will present the bitwise-OR, the bitwise exclusive-OR, and the bitwise inverse. Before we get to those episodes, let’s present what those operators are. The bitwise-OR operator is the vertical pipe, sometimes referred to as the vertical bar. You can find it on the US QWERTY keyboard as the shifted backslash key. The bitwise-XOR operator is represented using the caret or wedge, the symbol found above the 6 key on the standard US QWERTY keyboard. The bitwise-inverse, the unary operation that inverts all of the bits of an integer and does not use a mask, is represented with the tilde operator. You can find the key for this operator on the US QWERTY keyboard as the shifted key immediately to the left of the 1 key.

In our next episode, we will return to our discussion of the application of bitwise operators by examining the bitwise-OR. 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.