Episode 2.2 – Unsigned Binary Conversion

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.

In our last episode, we discussed how transistors can be used to represent binary numbers. In this episode, we will show how to go back and forth between decimal values and the computer's binary representation. We will also learn some binary tricks and show how to use those tricks in our software. Did you ever play that game where you try to figure out what the next number in a sequence is? Well, let's play it now. If I asked you to determine the next number in the sequence 1, 10, 100, 1,000, 10,000, you'd probably tell me that 100,000 is the next number. Yup, that sequence is based on the powers of ten. Each of these values in the sequence represents a different digit of the base-10 or decimal numbering system.

Now if I were to give you the sequence 1, 2, 4, 8, 16, and 32, you'd probably tell me that 64 is the next number, and you'd be right. This sequence is based on powers of two, and like the base-10 numbering system, each of these values represents a different digit of the base-2 or binary numbering system. There is a classic analogy for the relationship between decimal and binary that involves light bulbs. Assume you have eight light bulbs of different wattages. (In today's terminology, one would probably say "lumens" instead of "watts", but on we go.) The wattages of these hypothetical light bulbs are in increasing powers of two, and just like our sequence, they start with two to the zero-th power or 1 watt. With eight light bulbs, this would give us bulbs of 1 watt, 2 watts, 4 watts, 8 watts, 16 watts, 32 watts, 64 watts, and 128 watts. If each bulb is controlled with a different switch, we can have any combination of light bulbs on at one time and light the room with any wattage from zero up to all of light bulbs turned on.

Let's say we want to figure out which switches to flip to light the room with 200 watts. We start with the largest power of two that is less than or equal to our target wattage. That would be 128 watts, so we turn on that switch. That leaves us with 200 – 128 or 72 watts left to go.
From the remaining 72 watts, what is the next largest power of two we can pull out? It's 64. So we turn on the 64 switch which leaves us with 72 – 64 or 8 watts. Eight watts is less than 32 and 16, so we leave those switches off. Eight, however, is exactly equal to the next position, which is the two to the third power. So we turn on that bulb, which leaves us with 8 – 8 or 0 watts left to go. We now have exactly 200 watts on in the room.

Now let's record the positions of the switches using our binary representation. One hundred and twenty-eight is on, 64 is on, 32 is off, 16 is off, 8 is on, 4 is off, 2 is off, and 1 is off. If we record this as a binary number, we get that 200 in the base-10 numbering system is equal to 11001000 in base-2 or binary.

As an aside, it is possible for the number of bulbs to fall short of the desired wattage. There are three ways to tell if we do not have enough bulbs in the room. First, if the value left over after turning on the

largest wattage bulb (128 watts in the case of eight bulbs) is greater than or equal to the value of that bulb, we don't have had enough bulbs. The second way is to use the formula discussed during the last episode to calculate the largest possible value using the available bulbs. For n bulbs, that would be 2^n – 1. For eight bulbs, that would be 2^8 – 1 or 255. Therefore, any wattage over 255 watts is too much for our eight little bulbs. Lastly, if you perform the conversion with the bulbs that are available and did not get a remainder of zero, then you either made a mistake, or had too few bulbs.

This method of binary representation is called unsigned binary integer representation, or unsigned binary for short, and it is one of many different representations used by the computer. In unsigned binary, every binary digit or bit represents a different power of two.
Let's use an example to convert a binary value back to decimal. The 8-bit unsigned binary value 00101010 has ones in the positions assigned to $2^5 = 32$, $2^3 = 8$, and $2^1 = 2$. If we add these decimal values together, we get 32 + 8 + 2 = 42.

Okay, now it's time for those promised binary tricks. Let's take a look at six different binary to decimal conversions. First, the unsigned binary pattern 00000101 equals 4 + 1 = 5. If we move both of the ones in that unsigned binary pattern one position to the left, we get 00001010 which equals 8 + 2 = 10. Let's do this again. Moving the ones in the unsigned binary pattern one more position to the left gives us 00010100 which equals 16 + 4 or 20. Do this again and get 00101000 which equals 32 + 8 or 40. Repeat this, and get 01010000 or 64 + 16 which equals 80. And one last time to get 10100000 or 128 + 32 which equals 160.

Notice anything? Well, each time we moved the ones one bit position to the left, it doubled the result. Five changed to 10, 10 changed to 20, 20 changed to 40, and so on. There are many circuits inside a computer including units to perform addition, multiplication, division, and so on. Among these are circuits which allow the programmer to manipulate the individual bits of a number. This includes circuits that will take each binary digit of a value and move it different number of bit positions either right or left. When moving to the left, zeros are filled in from the right side. When moving to the right, either zeros or a copy of the leftmost or most significant bit are filled in from the left side. We will explain this difference in a later episode.

If you are familiar with a programming language such as Java or C, you have used arithmetic operators to perform operations such as addition, subtraction, multiplication, and division. Programming languages also have operators that allow us to access these bit manipulation circuits including a set of operators to perform bit shifts.

To shift bits to the left, we use the operator formed by two less-than signs next to each other, "<<". The element to the left of the operator represents the value or variable we wish to shift left, and the element to the right of the operator represents the number of bit positions by which we wish to shift the first value. For example, the instruction 5 << 2 would shift the binary pattern 00000101 two positions left to give us 00010100. This would be equivalent to multiplying 5 times 4.

Wait a moment. If a left shift by two bit positions is the same as multiplying by 4, why don't we just multiply by 4? Well, it has to do with speed. The circuitry used to perform the shift operation is much faster than the multiplication circuitry. If you have to perform many multiplications, the performance improvement found by using shifts could be significant. The benefit is so great that many processors don't even have a multiplication unit. The compilers simply figure out the best pattern of shifts to generate the correct result.

Okay, Tarnoff, you had me up to there. What if we want to multiply by something other than a power of two? Well, you simply combine the shifts with additions. Try this on for size. What would the result be for the expression (5 << 3) + (5 << 1) + (5 << 0). To begin with, 5 shifted left 3 times would equal 5 times 2^3 or 8. That gives us 40. Five shifted one position equals 5 times 2 or 10. Lastly, 5 shifted zero times simply equals 5. That would be 5 times the sum of 8 + 2 + 1, which equals eleven, which equals 55. Therefore, we used addition and shifting to multiply five times eleven.

Now let's shift the bits right. There are two operators for a right shift, >> and >>>. Later in this series, we will explain the difference. For now, it is enough to say that the syntax is the same as the left shift operator where the value to the left of the operator is the value or variable we wish to shift while the value to the right of the operator represents the number of bit positions we wish to shift the value right by. For example, the instruction 168 >> 3 would shift the binary pattern 10101000 three positions right to give us 00010101 or a decimal 21. This is equivalent to dividing 168 by 2^3 or 8 in order to get 21.
We can take advantage of the fact that a single shift one bit position to the right is equivalent to a division by two in order to fashion a method to convert a decimal integer to an unsigned binary integer. When you shift one bit position to the right, the rightmost bit has nowhere to go. For example, the binary representation for 5, 00000101, shifted one position right gives us 00000010 or 2 in decimal. Where did the least significant one go? Since we are sticking to integers for now, it dropped off as a remainder of one half.

As a number is shifted right one bit position at a time, each digit that is shifted out the right side is a record in reverse of the original binary value. That means that if you record the remainders generated by repeated divisions by two, you will find that after you've shifted right all of the way to zero, you've generated the binary equivalent of the original decimal value.

For example, let's convert the decimal value 156 to binary.

156 ÷ 2 = 78 with a remainder of 0
78 ÷ 2 = 39 with a remainder of 0
39 ÷ 2 = 19 with a remainder of 1
19 ÷ 2 = 9 with a remainder of 1
9 ÷ 2 = 4 with a remainder of 1
4 ÷ 2 = 2 with a remainder of 0
2 ÷ 2 = 1 with a remainder of 0
1 ÷ 2 = 0 with a remainder of 1

Listing the remainders in order with the last remainder as the most significant bit gives us 10011100, which is the binary value for the decimal integer 156.

In our next episode, we will take a look at a shorthand humans can use for binary and a shorthand computers can use for decimal. Until then remember that while the scope of what makes a computer is immense, it's all just ones and zeros.