Episode 3.06 – Fixed Point Binary Representation

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. Our topics involve a bit of figuring, so it might help to keep a pencil and paper handy.

In this episode, we're going to talk about representing real numbers in binary. For the purpose of our discussion, real numbers are all non-imaginary numbers including integers, fractions, and irrational numbers like pi or the square root of two. We've already shown a few ways to represent integers, but what about the rest of the real numbers? Can binary numbers have decimal points…er, uh, binary points?

It turns out that they can. To explain, we will once again start with decimal numbers. Remember that each digit of a decimal number represents a different power of ten with the power descending by one for each digit position that we move from left to right. A decimal point is added to separate the positions with non-negative exponents from the positions with negative exponents.

The first digit to the right of the decimal point represents 10^{-1} or one-tenth. As we move left to right, each subsequent position's value is one tenth of the one immediately to its left. That means that the decimal value 6.5342 is made up of 6 ones (10^0), 5 tenths (10^{-1}), 3 hundredths (10^{-2}), 4 thousandths (10^{-3}), and 2 ten-thousandths (10^{-4}).

Computers must also be capable of handling situations requiring more precision than integers can offer. Therefore, we need to be able to represent these fractional values in binary. Binary representation of fractional values works the same way as it does in decimal except that each position represents a power of two, not a power of ten. Now when we go from left to right, each position's value is one half of the one immediately to its left.

As for conversion, the process is the same as it was for unsigned binary integers except that the "wall" has been removed between the powers of two with non-negative exponents and the powers of two with negative exponents. If you listened to Episode 2.2, "Unsigned Binary Conversion," where we discussed conversion back and forth between decimal and unsigned binary, you can see that we're not presenting anything new here. The process is the same: break the decimal value into its powers of two components and place ones in the bit positions where powers of two exist and zeros where they don't. In the computer's hardware, where the number of bits are fixed, each bit represents a set power of two. By fixing the point within the available digits of a binary number where the exponents switch to negative values, we have fixed-point notation.

Let's convert the binary value 10.01101 to decimal. In this binary value, there are ones in the bit positions representing 2^1, 2^{-2}, 2^{-3}, and 2^{-5}. 2^1 equals 2, 2^{-2} equals one over four or 0.25, 2^{-3} equals one-eighth or 0.125, and 2^{-5} equals one over thirty-two or 0.03125. Add these powers of two together, and we get 2 + 0.25 + 0.125 + 0.03125, which equals 2.40625.

We also discussed in Episode 2.2 how shifting the bits of a binary number right is equivalent to dividing by powers of two while shifting the bits of a binary number left is equivalent to multiplying by powers of two. This works with fixed point values too, and in fact, it gives us another way to perform the conversion to decimal. Moving the binary point of 10.01101 five points to the right is equivalent to multiplying it by 2^5 or thirty-two. It also turns our fractional value into an integer: 1001101. Converting this integer to decimal gives us 64 + 8 + 4 + 1, which equals 77. We can now divide 77 by 32 to return us to the value represented by the original binary pattern. Seventy-seven divided by 32 is 2.40625.
Going from decimal to binary is going to reveal an interesting characteristic about the relationship between decimal and binary. For decimal values such as 0.5 or 0.25, the fractional powers of two are obvious. Things get a bit more difficult, however, with numbers like decimal 3.3. The integer powers of two we can remove from 3.3 are obvious: two plus one equals three. But how do we break the rest of this value, 0.3, into powers of two? Well, the largest power of two contained in 0.3 is 0.25 or one-quarter. Pulling 0.25 out of 0.3 leaves us with 0.05. The largest power of two in 0.05 is one thirty-second or 0.03125. Pulling one thirty-second out of 0.05 leaves us with 0.01875. We're not to zero yet, so we need to continue pulling out powers of two. The largest power of two in 0.01875 is 2^{-6} or 0.015625. Pulling this out of 0.01875 leaves us with 0.003125. We're still not to zero, so we go to the next lowest power of two that is less than what remains. 2^{-7} or 0.0078125 is too large, so we can't pull that out. 2^{-8} is still too large at 0.00390625. 2^{-9} is 0.001953125, which looks like the largest power of two we can pull out of 0.003125. This gives us 0.001171875.

Things don't look like they're getting any better, do they? Well, it turns out that they're not. The best we can do is to keep pulling out powers of two until we run out of bits representing negative powers of two. Eventually, we will bump up against the limits of accuracy for our binary system. If our system is limited so that the lowest power of two it can represent is 2^{-9}, then the best we can do to represent 3.3 in binary is 11.010011001. If we add up the powers of two represented by this fixed-point binary value, we get the decimal value 3.298828125. Close, but not quite exact. How do we get closer? Add more bits below the two to the negative ninth position.

By increasing our resolution all the way up to thirty-two bits with only two bits to represent the integers and the remaining 30 representing powers of two with negative exponents all the way down to 2^{-30}, we can represent 3.3 with 11.01001100110011001100110011001100, which equals 3.29999999981373548507690429687 5. Well, you get the idea, right? Regardless of how many binary digits we throw at representing a fixed-point value, we will never be able to exactly represent 3.3. This phenomenon has an effect on our computing. For example, how can we tell if a value equals 3.3 if we cannot represent 3.3? We will leave the answer to that question for a later episode.

In our next episode, we are going to start moving the binary point left and right with multiplication and division by powers of two in order to get scientific notation. Every number in scientific notation has parts including its sign, its fraction, and its exponent. We will use binary to represent these components of scientific notation in order to see how the computer stores a floating point value.
For transcripts, links, or other podcast notes, please check us out at intermation.com where you will also find links to our Instagram, Twitter, Facebook, and Pinterest pages. Until then remember that while the scope of what makes a computer is immense, it's all just ones and zeros.