Episode 2.4 – Packed BCD: Taking More Nibbles out of Binary

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 using hexadecimal as a shortcut for humans to represent binary. Turns out that computers have a problem too – they can't represent decimal. Okay, whole numbers like 42 don't pose a problem, but give them a number like 5.3, and they can't do it. Oh sure, they can get close. They'll only be off by a couple of tenths of millionths, but let's give 'em a break and provide a shortcut to accurate decimal representation. And like hexadecimal, this shortcut involves nibbles, you know, the groups of four bits that can take on one of sixteen possible patterns of ones and zeros.

So back to our shortcut. It turns out that there are some issues with using a base-two numbering system to represent decimal values. Well, actually there's just one, and until we cover fixed and floating point binary, it won't make much sense. For now it is enough to say that there are inaccuracies when we start trying to represent decimal non-whole numbers using binary. These inaccuracies are infinitesimal, but over many computations, they could accumulate a significant error. This is of particular concern with industries such as finance or manufacturing where the decimal units are expected to be precise.

The method of representation works like this. The computer uses separate groups of bits to represent each power of ten position in a decimal number. In other words, one group of bits represents just the decimal digit in the one's place, the next group of bits represents digit in the ten's place, the next group represents the hundred's place, and so on. This means that a number like 5,238 would require four of these bit groups: one to hold the 5, one to hold the 2, one to hold the 3, and one to hold the 8.
So how many bits are needed for each of these decimal places? Well, three bits won't be enough because they're only capable of representing eight different digits, and we need ten. The sixteen patterns of ones and zeros in a nibble, however, is more than enough. Using these four bits, we can represent 0 through 9, or in binary 0000 through 1001. Except in special cases, the six unused patterns from 1010 to 1111 should never appear in one of these bit groups.

If we use the two nibbles of a single eight-bit memory location, we can store two of these decimal digits. This means that an eight-bit memory location is capable of representing values from 0 to 99 by storing the tens digit in the first four bits and the ones digit in the last four bits. This mapping of two decimal digits to eight bits of binary is referred to as Packed Binary Coded Decimal, or Packed BCD.

Note that these 100 distinct decimal values don't use even half of the 256 patterns available in those same eight bits. That means that packed BCD offers much less storage density than unsigned binary. Some systems make this inefficiency even worse by using groups of six or even eight bits to represent each decimal digit. By the way, these alternate forms of BCD pose a problem. Without a single standard, different systems could use different variants.

Later in this series, we will discuss how 10's complement representation can be used to represent negative values in decimal, and consequently, in BCD too. An early alternative for representing negative numbers in BCD, however, was to set aside a nibble to contain either a plus sign or a minus sign. Typically, the nibble reserved for the sign was the last or rightmost nibble, a contradiction to the way in which humans place the plus or minus sign at the leftmost position. A binary 1100 (a C in hexadecimal) was typically used to represent a plus sign while a binary 1101 (a D in hexadecimal) was typically used for the minus sign.

Before we end this episode, let's do a quick example by converting 5,238 to Packed BCD. This four digit decimal number will require four nibbles or two bytes in binary. The first digit, 5, in a binary nibble is 0101, while 2 is 0010, 3 is 0011, and 8 is 1000. So in Packed BCD, 5,238 would be 0101 0010 0011 1000. If we were to include a nibble for the sign, we would need five nibbles with a 1100 representing the plus sign added to the right side. Since a byte contains two nibbles, we'd need to have an even number of nibbles. We fix this by adding a leading zero to the left side. This would give us 0000 0101 0010 0011 1000 1100.

By the way, it turns out that there is another advantage to using BCD. The simplicity of conversion between BCD and decimal makes it a method to consider when displaying decimal values such as on clocks and calendars. It is for that reason that many computer clocks, often referred to as real-time clocks, use BCD. Assume you're going to store hours, minutes, seconds, and hundredths of seconds in separate memory locations. Since each time element is getting its own eight-bit memory location, and since none of these elements exceeds 99, then why not use BCD? It simplifies the conversion.

In our next episode, we are going to examine yet another use of binary numbers – the digital representation of analog measurements. At first glance, it seems like we would just employ a straight decimal to binary conversion to do this, but we don't. An understanding of the relationship between the range of analog values and the number of bits used in the digital values will help us understand accuracy and conversion. It will also explain why computers come up with some of the funny values that they do when representing things such as position, brightness, temperature, and acceleration. Until then remember that while the scope of what makes a computer is immense, it's all just ones and zeros.