Episode 2.3 – Hexadecimal or Sixteen ways to nibble at 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 the basics of converting binary to decimal and vice versa. We also presented some binary tricks including one that allows us to multiply integers using only bit shifts and addition. This week, we're taking a look at a shorthand humans can use to quickly represent binary. Quick, copy down this 16-bit number: 1010111001111100. Whew! Did you get it? Eh, probably not. There are a few challenges to having only the two numerals, zero and one, in binary representation. First, binary values tend to have a lot of bits. For example, the decimal value that corresponds to that 16-digit binary value given earlier takes less than a third of those digits to be represented in decimal. Everyday integers such as 2019 become rather unruly when writing them out in binary. The four-digit decimal value 2019, by the way, requires a minimum of 11 bits in unsigned binary and is equal to 0111 1110 0011.

A second problem is that long repeating strings of ones, like the six ones in a row we found in the binary representation of 2019, or likewise long repeating strings of zeros could be difficult for a human to accurately recognize or duplicate. An alternate representation of binary that utilizes more than just the two symbols would help with this. And lastly, quick recognition of the relative magnitude of two binary values does not come naturally.

So, to come up with our shortcut, let's begin by partitioning the binary number into groups of, say, three bits. For example, the 16-bit binary value I asked you to copy down earlier can be divided into 5 groups of three bits plus a single bit left over at the beginning of the number. These groups would be 1 followed by 010 followed by 111 followed by 001 followed by 111 followed by 100. Already we can see that partitioning the number has made it easier to copy. It is important to note here that we always start this partitioning from the right or least significant end of the number so that any partial group of bits ends up on the left or most significant end.

Now each of these groups of three bits can take on 2^3 or eight different patterns of ones and zeros. What we need now is eight different numerals to represent each of these eight patterns. The easiest solution is to map each three-bit pattern to its corresponding decimal value. For example, three zeros equals 0, 001 equals 1, 010 equals 2, 011 equals 3, and so on. By doing this, our 16-bit binary value can be represented with 6 digits: 127174. This new representation is both easier to use and takes less digits to write.

It turns out that what we've done is converted our binary value to a base-8 or octal value where each digit represents a different power of eight. Because of the way the partitions divide the binary number at the same places that an octal value separates its digits, we can quickly convert a binary representation to octal.

Going the other direction, from octal to binary, is just as simple. By substituting each octal digit with its corresponding three-bit binary equivalent, we can convert any octal number to binary. For example, the octal value 27531 is equivalent to the binary value 010 111 101 011 001 where the 2 equals 010, the 7 equals 111, the 5 equals 101, the 3 equals 011, and the 1 equals 001. Remember not to delete any leading zeros from any of the groups of bits. They are needed in order to make sure the digits line up with the appropriate magnitude for their bit position.

You may have noticed that partitioning our 16-bit binary value generated a leftover bit at the leftmost part of the number. This is because sixteen is not divisible by three. Early computing systems such as those commercially available in the 1960's and 70's used 6, 12, 24, or 36 bits for their binary values. Each of these bit widths is easily divisible by 3, so octal was an efficient shortcut to represent their binary values.

For reasons that will become clear later in this series, modern computing systems represent values with bit widths that are powers of two. The typical memory location, for example, is capable of storing eight bits. That means that all values stored to memory will be a multiple of 8 bits wide. Since eight is not divisible by three, it might be nice to choose another way to partition our binary values for this shortcut. Two-bit partitions would not give us much of an advantage over binary, so let's partition our number into groups of four bits. As a side note, remember that a group of four bits can be referred to as a nibble.

Now as we did with octal, we can use a single numeral to represent each of the possible patterns of ones and zeros found in a nibble. The first ten patterns of ones and zeros in a nibble are easy: just use the decimal value that is equivalent to the binary value. In other words, the nibble 0000 is just 0, 0001 is 1, 0010 is 2, 0011 is 3, and so on all the way up to 1001, which equals 9.

Unfortunately, we cannot continue to use decimal to represent the remaining six patterns of ones and zeros in a nibble. That would make it impossible to distinguish whether 12 represented the single nibble 1100 or the two nibbles of 0001 followed by 0010. So instead, we begin using letters. The nibble after 9, 1010, is assigned to A, 1011 is B, 1100 is C, 1101 is D, 1110 is E, and 1111 is F. These represent the decimal values 10, 11, 12, 13, 14, and 15 respectively.

This time what we've done is converted our binary value to base-16 or hexadecimal where each digit represents a different power of sixteen. Because of the way the partitions divide the binary number at the same places that a hexadecimal value separates its digits, we can quickly convert a binary representation to hexadecimal.

So, let's go back to our 16-bit value. First, we need to partition that number into nibbles. This gives us 1010 1110 0111 1100. Next, we need to replace each nibble with its corresponding hexadecimal digit. We replace 1010 with A, 1110 with E, 0111 with 7, and 1100 with C. This gives us the hexadecimal value AE7C. These four hexadecimal digits are much easier to work with than the sixteen ones and zeros of the original binary value. In addition, we can quickly see that AE7C is less than AE9C, a task that might not be so easy in binary.

Converting from hexadecimal to binary is as simple as substituting the hexadecimal digit with its corresponding four-bit binary equivalent. For example, the hexadecimal value 5D3F219A is equivalent to the binary value 0101 1101 0011 1111 0010 0001 1001 1010. Well that was quick. And just as with our

octal to binary conversion, do not delete any leading zeros. They are needed in order to make sure the digits line up with the appropriate magnitude for their bit position.

Oh, and one last thing about hexadecimal. Computers do not use hexadecimal, humans do. Hexadecimal provides humans with a reliable shorthand for writing large binary numbers. It would be nice, however, to be able to use hexadecimal in our computer programs. Funny you should say that. In most programming languages, a hexadecimal value is distinguished from a decimal value by putting a zero followed by a lowercase x directly in front of the hexadecimal value. AE7C, for example, would be written as 0xAE7C with no spaces.

Now that we've seen how hexadecimal can serve humans as a shorthand for binary, we will spend our next episode examining a shorthand for computers to represent decimal. Until then remember that while the scope of what makes a computer is immense, it's all just ones and zeros.