Learning the Binary System

WWW.BRAINLUBEONLINE.COM/COUNTING_ON_BINARY/

Formated to LaTeX: 11/25/2012

Abstract

This is a document on the base-2 abstract numerical system, or Binary system. This is a VERY easy concept to learn, it's amazing that the binary system is not common knowledge! In fact it's easier than the system you already know!

Contents

1 What is it?

We should begin by defining a *numeral.* A *numeral* is a symbol that describes a number. Expressions such as: *4*, *IV*, *78*, *101* are all *numerals* describing numbers. A *numeral* is not a number, it is an expression describing a number. A number is a concept of quantity. One must know the difference because the same number may be described by many different numerals. An example would be 4 (in radix-10, or Base-10, the way we think), IV(in Roman numerals), or (100, in Radix-2, or Base-2, otherwise known as Binary). The fact is, for small numbers, it is actually easier to think in base-2, or binary, than it is to think in base-10, or decimal. Multiplication and division is way easier in binary than the decimal way we were taught in school.

Thinking in different bases could be compared to a farmer looking into egg cartons, counting by 12s; more on this later. What this means is 3 dozen (base-12) is 3*12 = 36, in decimal (base 10, not base 12) that would be 3*10 + 6 = 36. Just keep reading, you'll get it, really! Radix n is a numeration system with a base of n. Later we will learn about octal numbers (a numeration system with a base of 8, or radix 8 where the *n* in *Radix n* is 8).

A clock uses a Radix 60, numeration system, at least the seconds and minutes. These are just good examples to get you thinking about the concept, and difference between *numbers* and *numerals.* A quick example of a an octal number would be 26_8 The eight in subscript tells us we are looking at a number described in Radix-8 notation. We will get into octals later, you do not have to understand this right now, this is just to introduce you to the concept. This number represents 2*8+6 in base-10 or 22_{10}. This will make sense later.

When we write a number we assume it is in decimal format (radix-10), unless the subscript tells us otherwise, or it is obviously implied. This means 14_{10} is unnecessary because we already will assume that the text means 14 in decimal.

Base-2, or the binary system is the simplest system one can count in because there are only two symbols to indicate a *number* 1 & 0 that's it. The decimal system needs 10 different symbols to describe the same numbers 0-9. and each place from the right to the left is a multiple of 10.

10 to the Power of: (10^n)	10^{6}	10^{5}	10^{4}	10^{3}	10^{2}	10^{1}	10^{0}
	1,000,000	100,000	10,000	1,000	100	10	1

This means that the place from right to left, determines the power of the numeral. The same is true for the binary system.

2 to the Power of: (2^n)	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
	64	32	16	8	4	2	1

That's all there is to it. Now let's try converting some numbers from decimal to binary.

First lets try 48.

Decimal:	128	64	32	16	8	4	2	1
Binary:	0	0	1	1	0	0	0	0

32 + 16 = 48, this is 110000 in binary, read from right to left. This is six characters long (the most significant bit is six from the right) so this would be at it's smallest, a six *bit* number. Usually, computers reserve a set amount of memory for numbers in sectors with an equal size, such as 8, 16, 32, or even 64 bits. So this number in an 8-bit register would be 00110000 (in other words the extra zeros to the left are put in as *padding*, this will be important later when we talk about bit shifting, but for now don't worry about it.)

Let's do another one, how about 19.

Decimal:	128	64	32	16	8	4	2	1
Binary:	0	0	0	1	0	0	1	1

This gives us the 8-bit binary number 00010011. This is because the largest power of 2 (16 in base-10) is the most significant bit, the remainder is 3_{10}, this means 2^1 will go into that once with a remainder of 1_{10} which is 2^0. (Remember from a few tables above that $2^0 = 1_{10}$, $2^1 = 2_{10}$, and $2^2 = 4_{10}$)

2 Why use it? Why not?

It is thought that we use the decimal system because we have 10 fingers, but the truth is we could have just as easily used the binary system naturally. The Mayans used a base-20 system, perhaps because they used their toes also. The reason we want to be able to think in binary is for programming and circuit design. The most basic TTL component is the *flip-flop* (a switch). It is High or Low, On of Off, 1 or 0. If we wanted a computer to think in base-10 we would need 9 different states not including a 0. With Radix-2 we only need 2 states, 1 or 0. This is the most efficient numeration system a digital computer can work with at the lowest level. As a matter of fact this will remain true for ALL computers until analog, or multi-state computers are invented.

The question is; if it's so great, why don't we use it all the time? Well it can get very tedious to write, for instance 36771 in decimal notation only takes 5 characters to describe. In binary notation it is: 1000000000000011, the same number takes 16 characters to describe, over 3 times as many characters! The reason is simple, decimal notation has 10 characters to choose from to describe a number 0-9, binary only has 2 characters, 1 and 0. There are systems that can shorten decimal numbers also, such as base-16, known as the hexadecimal system that we will learn about later. The same number in hex is 8003, this is because in hex notation there are 16 characters to choose from, 0-F! For instance 3A in hex is equal to 58 in decimal. This is because A is in the 1's column (16^0), and 3 in the 16's column (16^1) so (3*16+10 = 58, 10 = A in hex, carry the 1 in decimal), more on that later.

To fully understand binary, we will need to study another type of numeration system as well, this system is referred to as the octal system. Yes you guessed it, the octal system is Radix-8, using the characters 0 though 7. The reason we need to learn about counting in base-8 will become clear later. Working in octal numbers is an easy way to convert binary numerations to decimal notation, octal numbers are sort of a mid point. Every three bits of a binary number represent an octal digit. This makes it easy to look at a binary number and see what it is as an octal. Again more on this later, the writing here is just to tell you what is coming so you are not thrown into the world of unknown numeral systems.

3 Binary Numbers

3.1 Binary Conversion

OK, so now we have a vague understanding of binary notation. The next question is; do we have an understandable, programmable way of getting binary numbers from decimal numbers? The answer is yes! Lets start with 7_{10}; all you have to do is keep dividing by 2 until there is no number left. I'll explain, the first divide by 2, is 3 in decimal. This gives us a remainder of 1_{10}; there are only two numbers a remainder can be in binary or when dividing by two, 1 or 0. This is a powerful piece of information because it makes all your calculations much easier. When you divide to get binary, you only record the remainders, and you only operate (divide) on the result of your division. This will become clear in a moment, this table will help you see how to convert 7 in decimal to 0111 in binary.

Divide:	$7/2 = 3$	$3/2 = 1$	$1/2 = 0$
Remainder:	$= 1$	$= 1$	$= 1$

You than flip the result and that gives you 111, or 0111 with a zero on the left to pad the number to 4 bits.

Now lets try converting 9_{10} to binary.

Divide:	$9/2 = 4$	$4/2 = 2$	$2/2 = 1$	$1/2 = 0$
Remainder:	$= 1$	$= 0$	$= 0$	$= 1$

Notice that each time we divide (9/2 gives us 4, then we divide 4 by 2, that gives us 2, then 2/2 give us 1 and so forth) we only record the remainders. Then, we flip the results, and there is our number 1001_2. Up until now our numbers have been symmetrical so the flip has not been apparent. Lets try one that is not symmetrical:

Let's try 23_{10}.

Divide:	$23/2 = 11$	$11/2 = 5$	$5/2 = 2$	$2/2 = 1$	$1/2 = 0$
Remainder:	$= 1$	$= 1$	$= 1$	$= 0$	$= 1$

The answer is 10111_2. Now it matters which way we read the number, 11101 is a very different number than 10111. We always start reading the binary number from the last operation we worked on (bottom to top). That's it, that is the hardest part, it's all down hill from here!

Converting binary numbers to decimal notation is as easy as adding. The 8-bit number 00010011_2 for example.

Decimal:	128	64	32	16	8	4	2	1
Binary Place:	0	0	0	1	0	0	1	1

Just add where you see a 1. (16 + 2 + 1 = 19)

Just to review, for any number denoted a, where a is any non-zero number:

a^0	a^1	a^2	a^3	a^4	a^n
Always = 1	Always = a (the value of a)	$=a*a$	$=a*a*a$	$=a*a*a*a$	$=a*a...n$ times

3.2 Addition

To add two numbers in binary, you do it the same way as you do in the decimal system, only it is much easier in binary. This is because you are only adding, or carrying, a 1, and the result in each column can only be a 1, or a 0. So here we go:

$$\begin{array}{rr} & 0 \\ + & 0 \\ \hline & 0 \end{array}$$

$$\begin{array}{rr} & 1 \\ + & 0 \\ \hline & 1 \end{array}$$

$$\begin{array}{rr} & 1 \\ + & 1 \\ \hline 1 & 0 \end{array}$$

A ’*1*’ is carried

$$\begin{array}{rr} & 1 \\ & 1 \\ + & 1 \\ \hline 1 & 1 \end{array}$$

A ’*1*’ is carried and a ’*1*’ remains

$$\begin{array}{rrr} & & 1 \\ & & 1 \\ & & 1 \\ & + & 1 \\ \hline & 1 & 0 \\ + & 1 & 0 \\ \hline 1 & 0 & 0 \end{array}$$

Above we just turn the 4 numbers into 2 (we add the top 1 + 1, then the bottom 1 + 1) Then we simply add those two binary numbers together as well.

$$\begin{array}{cccccccc} & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ + & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ \hline 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{array} \quad \text{In decimal this is:} \quad \begin{array}{ccc} & 9 & 1 \\ + & 5 & 8 \\ \hline 1 & 4 & 9 \end{array}$$

The last one here, is not realistic because the two numbers to be added were 7 bits long. Let's pad them (on the left of course so it doesn't change their value to make them 8 bits long).

$$\begin{array}{ccccccccc} & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ + & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ \hline & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{array} \quad \text{In decimal this is:} \quad \begin{array}{ccc} & 9 & 1 \\ + & 5 & 8 \\ \hline 1 & 4 & 9 \end{array}$$

All we did was add a zero to each number to bring it up to 8 bits in length. If we were working on a system where the registers WERE actually 7-bit, then we would have a problem because the result of the addition is 8 bits in length and the last bit (to the left) would be truncated (lost). The computer would see this if it had 7-bit registers:

$$\begin{array}{cccccccc} & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ + & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ \hline & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}$$

In decimal this is: $\begin{array}{ccc} & 9 & 1 \\ + & 5 & 8 \\ \hline & 2 & 1 \end{array}$ **!˜˜ WRONG ˜˜!**

3.3 Subtraction

Subtraction in binary is the same as in decimal. You borrow just like you would in the decimal system the only difference is that you are just working with 1 and 0. Here we go:

$$\begin{array}{cc} & 0 \\ - & 0 \\ \hline & 0 \end{array} \quad \text{Binary} \quad \begin{array}{cc} & 0 \\ - & 0 \\ \hline & 0 \end{array} \quad \text{Decimal}$$

$$\begin{array}{cc} & 1 \\ - & 0 \\ \hline & 1 \end{array} \quad \text{Binary} \quad \begin{array}{cc} & 1 \\ - & 0 \\ \hline & 1 \end{array} \quad \text{Decimal}$$

$$\begin{array}{rr} & 1 \\ - & 1 \\ \hline & 0 \end{array} \text{ Binary } \qquad \begin{array}{rr} & 1 \\ - & 1 \\ \hline & 0 \end{array} \text{ Decimal}$$

$$\begin{array}{rrr} & 1 & 0 \\ - & 0 & 1 \\ \hline & & 1 \end{array} \text{ Binary } \qquad \begin{array}{rr} & 2 \\ - & 1 \\ \hline & 1 \end{array} \text{ Decimal}$$

You must remember, whenever you borrow, you are borrowing from two times the amount (one column to the left). Go over the following carefully. Here is an example of borrowing:

$$\begin{array}{rrrrrr} & 1 & 0 & 1 & 1 & 0 \\ - & 0 & 1 & 0 & 1 & 0 \\ \hline & 0 & 1 & 1 & 0 & 0 \end{array} \text{ Binary } \qquad \begin{array}{rrr} & 2 & 2 \\ - & 1 & 0 \\ \hline & 1 & 2 \end{array} \text{ Decimal}$$

$$\begin{array}{rrrrrr} & & 1 & & & \\ & 1 & 1 & 1 & 1 & 0 \\ - & 0 & 1 & 0 & 1 & 0 \\ \hline & 0 & 1 & 1 & 0 & 0 \end{array}$$

Remember When Borrowing: It is the same as putting two ones over the borrowing column.

3.4 Multiplication

Multiplication and division will never get easier than this, because in binary you will never multiply (or divide) anything greater than 1! Multiplication is performed with the same method used in decimal mathematics, from right to left, and shifting left with each new operation multiplier.

$$\begin{array}{rrrrr} & & 0 & 1 & 1 \\ & * & 1 & 1 & 0 \\ \hline & & 0 & 0 & 0 \\ & 0 & 1 & 1 & \\ 0 & 1 & 1 & & \\ \hline 1 & 0 & 0 & 1 & 0 \end{array} \text{ Binary } \qquad \begin{array}{rrr} & & 3 \\ * & & 6 \\ \hline & 1 & 8 \end{array} \text{ Decimal}$$

$$\begin{array}{ccccccccccccccc}
 & & & & & & & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0\\
 & & & & & * & & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1\\
\hline
 & & & & & & & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0\\
 & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
 & & & & & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & & \\
 & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & \\
 & & & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & & & & \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & \\
1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & & & & & & & \\
\hline
1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0
\end{array}$$

Decimal

Binary

$$\begin{array}{ccccc}
 & & 2 & 1 & 0\\
 & * & 1 & 4 & 9\\
\hline
 & 1 & 8 & 9 & 0\\
 & 8 & 4 & 0 & \\
2 & 1 & 0 & & \\
\hline
3 & 1 & 2 & 9 & 0
\end{array}$$

You should notice the pattern above. When multiplying by a 0, it is zero across the board, when multiplying by 1, it is the same number it always is (the top number). This is important when we get into bitshifting. With a large enough register, we can just shift and add to multiply. For instance if we count one set of zeros, then a 1's row again we shift twice to the left then add, if we count two rows of zeros then a 1's row again, we shift 3 times to the left than add. You do not have to understand this right now. Really, if you understand multiplication in binary, then this page has done it's job so far. Hang with it!

3.5 Division

Division is just as simple in binary notation becuase when you divide, your numbers may either go into the number, or not. In other words if the divisor goes into the number you are dividing, it will only go in once, or not at all. The only quotent you can have is either a 1 or a 0.

1000_2 œ $100_2 = 10_2$ 8_{10} œ $4_{10} = 2_{10}$

$$\begin{array}{cccccccc}
 & & & & & & 1 & \\
\cline{5-8}
1 & 0 & 0 & / & 1 & 0 & 0 & 0\\
 & & & & 1 & 0 & 0 & \\
\cline{5-8}
 & & & & 0 & 0 & 0 & 0
\end{array}$$

You just divide and subtract just like you would with the decimal system. Here is a larger example:

$$\begin{array}{ccccccccccc}
\cline{6-11}
1 & 0 & 0 & 1 & / & 1 & 1 & 0 & 1 & 1 & 0
\end{array}$$

$$\begin{array}{ccccccccccc}
 & & & & & & & & 1 & & \\
\cline{6-11}
1 & 0 & 0 & 1 & / & 1 & 1 & 0 & 1 & 1 & 0\\
 & & & & & 1 & 0 & 0 & 1 & & \\
\cline{6-11}
 & & & & & 0 & 1 & 0 & 0 & 1 & 0
\end{array}$$

$$\begin{array}{ccccccccccc}
 & & & & & & & & 1 & 1 & \\ \cline{6-11}
1 & 0 & 0 & 1 & / & 1 & 1 & 0 & 1 & 1 & 0 \\
 & & & & & 1 & 0 & 0 & 1 & & \\ \cline{6-11}
 & & & & & 0 & 1 & 0 & 0 & 1 & 0 \\
 & & & & & & 1 & 0 & 0 & 1 & \\ \cline{6-11}
 & & & & & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

$$\begin{array}{ccccccccccc}
 & & & & & & & & 1 & 1 & 0 \\ \cline{6-11}
1 & 0 & 0 & 1 & / & 1 & 1 & 0 & 1 & 1 & 0 \\
 & & & & & 1 & 0 & 0 & 1 & & \\ \cline{6-11}
 & & & & & 0 & 1 & 0 & 0 & 1 & 0 \\
 & & & & & & 1 & 0 & 0 & 1 & \\ \cline{6-11}
 & & & & & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & 0 & 0 & 0 & 0 & 0 & 0 \\ \cline{6-11}
 & & & & & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

4 Bitwise Operations

4.1 1's Compliment

TTL circuits generally deal with multiplication by adding the number to be multiplied, to itself n number of times. In other words 4 * 9 is adds 4, 9 times. Division is the same way, the circuits just use subtraction instead of addition as division is the inverse of multiplication. Processors can be a little different but usually the compiler will shield you from having to think about these things. This brings us though, to subtraction. Computers do not really need to subtract, only add by an operation called *1's compliment* to get a subtracted result. When performing subtraction, the number you are subtracting numbers from is called the *minuend*. The amount you are subtracting is called the *subtrahend*.

Now that we have a few of the terms out of the way let's start talking about 1's compliment. 1's compliment is where you turn the 1's in a binary number into 0's and the 0's into 1's, just write the exact *inverse* of whatever state that column in the number reads. For example 10011010 is 01100101 in 1's complement. To subtract in binary you simply invert the subtrahend and add to the minuend. To better put it, you add the minuend to the 1's complement of the subtrahend. If the sum of the addition adds a digit, we carry this (a 1_2) to the right-side end. This is called *end-around carry* here are some examples:

Remember: When borrowing in n_2 (Binary), it is the same as putting two ones over the borrowing column.

$$\begin{array}{cccccc} & 1 & 0 & 1 & 1 & 0 \\ - & 0 & 1 & 0 & 1 & 0 \\ \hline & & 1 & 1 & 0 & 1 \end{array} \quad \text{or} \quad \begin{array}{rccccc} & 1 & 0 & 1 & 1 & 0 \\ + & 1 & 0 & 1 & 0 & 1 \\ \hline \text{End-around Carry 1:} & 0 & 1 & 0 & 1 & 1 \\ & & & & & 1 \\ \hline & & 1 & 1 & 0 & 0 \end{array}$$

Remember: add the *end-around carry.*

This works because by adding the inverse you are dealing with what's left as a value from 0. If you think about it for a while it will become clearer.

If there is not a 1_2 to carry, we have a negative result. If you have a negative result, you will need to recomplement the result (invert it). This is important because you will not get the result you are looking for if you subtract in the standard fashion and you get a negative number:

$$\begin{array}{cccccc} & 1 & 0 & 1 & 1 & 1 \\ - & 1 & 1 & 0 & 1 & 0 \\ \hline & - & 1 & 1 & 0 & 1 \end{array} \quad \textbf{WRONG!}$$

or

$$\begin{array}{rccccc} & 1 & 0 & 1 & 1 & 1 \\ + & 0 & 0 & 1 & 0 & 1 \\ \hline \text{Invert:} & 1 & 1 & 1 & 0 & 0 \\ \hline - & 0 & 0 & 0 & 1 & 1 \end{array} \quad \textbf{Correct!}$$

This happens because the computer (or human) will effectively borrow to infinity. If you want to get around this and still use the standard human method of binary subtraction just make your subtrahend the smallest number and know that that result is a negative.

With the micro-controllers available now, you will rarely need to deal with TTL circuits and the compilers now are so advanced you really will probably not need to deal with 1's complement. You will deal with 2's complement however. 2's complement is insanely easy to explain. The last bit is inverted for negative numbers. That's it, that is all there is to it. This means an 8-bit register that would normally go from 0_{10} (00000000_2) to 255_{10} (11111111_2), would have 7 bits (one less) for a magnitude and one bit reserved for a polarity. Using the 2's complement syntax that register will now describe -127_{10} (10000000_2) to 127_{10} including a 0_{10}. The -0_{10} takes a little thinking about, you are not gaining any room, the 0 has the potential to be it's inverse just as the rest of the numbers do. It will become clear with a little pondering.

4.2 Big and Little Endians

Endiannism there is an interesting etymology for this word that is beyond the scope of this page. We will however discuss the terms meaning. A packet is a block on information, they can vary in size, for our purposes we will make our packet size 1 byte (8-bits). The Endianness of a packet of information, a byte (anything 8-bits in length) for example is the description of which way the data travels. For instance, in these articles we have been reading binary numbers with the least significant bit on the right. A computer may see things differently, a computer may read information in a little-endian way or a big-endian way. Here is a table to help us:

Endian	First Bit	Last Bit
Little	Least Significant	Most Significant
Big	Most Significant	Least Significant

Early Motorola and IBM chips were big-endian and Intel chips were little-endian. This means that the IBM chips interpreted the first bit of an 8-bit variable as the 128 column. In reality there are many different types of endians, for instance with large packets of packets. Another example are systems that treat a 32-bit number as two 16-bit numbers. This really needs to be dealt with on a project to project basis. I mention it here only as a diagnostic step you may want to take if you are having trouble.

5 Bit Shifting

Bitshifting is the practice of shifting all the columns of a binary number to the left or the right. The left-shift operator in languages with a C like syntax look like this << . The left-shift operator will shift the bits to the left in a variable *n* times, with a default shift of 1 place. This means 00001011 shifted to the left once will now be 00010110. This effectively doubled the numbers magnitude. What happens with a number like 1011 though? Shifted once to the left, this number would end up 0110. This is because the left-shift operator is not an end-around carry operator. The original number is lost. When using the left-shift operator, you may shift 1, 2,...n times to the left by the syntax *N* << *timesToShift*; where N is the variable to be shifted and timesToShift is the, you guessed it, the # of times the variable is to be shifted.

# To Shift:	Not Shifted	*N*<<1	*N*<<2	*N*<<3
Number	10101100	01011000	10110000	01100000

This brings us the the right-shift. >>> The right-shift operator does the inverse of the left-shift. 101100 shifted once to the right would be 010011 all information shifted off the right is lost. The right-shift operator divides the number, if you right-shift once you have divided the number in half. If the number was an odd number, you are truncating the modulus.

# To Shift:	Not Shifted	*N*>>>1	*N*>>>2	*N*>>>3
Number	10101100	01010110	00101011	00010101

Ok, so what happens when we are using 2's compliment and trying to bit-shift in this manor? We get funny results! They've already thought about this, the operator is the arithmatic-right-shift operator. >> This operator is not an end-around carry operator, but it does pad with the state of the most significant bit (the polarity bit). For instance this number: 10001110 shifted right arithmetically 3 times would be 11110001. There is not an arithmatic-left-shift operator because it would not produce a number you could not derive from the operators already given.

# To Shift:	Not Shifted	N>>>1	N>>>2	N>>>3
Number	10101100	11010110	11101011	111110101

# To Shift:	Not Shifted	N>>>1	N>>>2	N>>>3
Number	00101100	00010110	00001011	00000101

6 Other Computer Numbering Systems

6.1 Octal System

Decimal Number	0	1	2	3	4	5	6	7	8	9
Octal Number	0	1	2	3	4	5	6	7	10	11

Decimal Number	10	11	12	13	14	15	16	17	18	19	20
Octal Number	12	13	14	15	16	17	20	21	22	23	24

The octal system is radix-8 (N_8), this system uses the numerals 0-7. This system is a great thing to know. When you're using the octal system, you can at a glance, know what number you are looking at in binary. Every three binary columns are an octal number. Let's look at a table:

Binary Number	Decimal Equivalent	Octal Equivalent
101101	$32+8+4+1=45_{10}$	$55_8=5*8+5=45_{10}$
111100	$32+16+8+4=60_{10}$	$74_8=7*8_{10}+4=45_{10}$

Every three binary columns can go from 0 - 7 in magnitude. This means you are either staring at a 1's column, a 2's column, or a 4's column. Every set of three to the left over you go, you add a * 8. (C^{n}).

This table will show Octal numbers:

Binary Number	Octal Sets	Conversion Equivalent
10110101	10 110 101	$2*8^2{}_{10}+6*8^1{}_{10}+5*8^0{}_{10}=181_{10}$

6.2 Hexadecimal System

The Hexadecimal system, hex, is radix-16. Until now, we have dealt with numeration systems with less characters than the decimal system. We are now going to look at a system with 16 different glyphs with magnatudes from 0 to 15. Instead of a subset showing the base, in hex we simply add 0x as a prefix to show that we are working in hex:

Decimal Number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Hex Number	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17

Here are some larger numbers:

Decimal Number	40	255	256	257	89372	1512394
Hex Number	0x28	0xFF	0x100	0x1001	0x15D1C	0x1713CA