

Why Binary?

The number system that you are familiar with, that you use every day, is the *decimal number system*, also commonly referred to as the *base-10* system. When you perform computations such as $3 + 2 = 5$, or $21 - 7 = 14$, you are using the decimal number system. This system, which you likely learned in first or second grade, is ingrained into your subconscious; it's the natural way that you think about numbers. Of course it is not just you: It is the way that everyone thinks—and has always thought—about numbers and arithmetic. Evidence exists that Egyptians were using a decimal number system five thousand years ago.

The Roman numeral system, predominant for hundreds of years, was also a decimal number system (though organized differently from the Arabic base-10 number system that we are most familiar with). Indeed, base-10 systems, in one form or another, have been the most widely used number systems ever since civilization started counting.

In dealing with the inner workings of a computer, though, you are going to have to learn to think in a different number system, the *binary number system*, also referred to as the *base-2* system.

Before considering why we might want to use a different number system, let's first consider: *Why do we use base-10?* The simple answer: We have 10 fingers. Before the days of calculators and computers, we counted on our hands (many of us still do!).

Consider a child counting a pile of pennies. He would begin: “One, two, three, ..., eight, nine.” Upon reaching nine, the next penny counted makes the total one single group of ten pennies. He then keeps counting: “One group of ten pennies... two groups of ten pennies... three groups of ten pennies ... eight groups of ten pennies ... nine groups of ten pennies...” Upon reaching nine groups of ten pennies plus nine additional pennies, the next penny counted makes the total thus far: one single group of one hundred pennies. Upon completing the task, the child might find that he has three groups of one hundred pennies, five groups of ten pennies, and two pennies left over: 352 pennies.

More formally, the base-10 system is a *positional* system, where the rightmost digit is the ones position (the number of ones), the next digit to the left is the tens position (the number of groups of 10), the next digit to the left is the hundreds position (the number of groups of 100), and so forth. The base-10 number system has 10 distinct symbols, or digits (0, 1, 2, 3,...8, 9). In decimal notation, we write a number as a string of symbols, where each symbol is one of these ten digits, and to interpret a decimal number, we multiply each digit by the power of 10 associated with that digit's position.

For example, consider the decimal number: 6349. This number is:

$$\begin{array}{ccccccc} \underline{6} & \underline{3} & \underline{4} & \underline{9} & = & 6 \times 10^3 & + 3 \times 10^2 + 4 \times 10^1 + 9 \times 10^0 \\ \uparrow & \uparrow & \uparrow & \uparrow & & & \\ | & | & | & | & & & \\ 10^3 & 10^2 & 10^1 & 10^0 & & & \\ \text{position} & \text{position} & \text{position} & \text{position} & & & \\ (\text{i.e., thousands position}) & (\text{i.e., hundreds position}) & (\text{i.e., tens position}) & (\text{i.e., ones position}) & & & \end{array}$$

There is nothing essentially “easier” about using the base-10 system. It just seems more intuitive only because it is the only system that you have used extensively, and, again, the fact that it is used extensively is due to the fact that humans have 10 fingers. If humans had six fingers, we would all be using a base-6 system, and we would all find that system to be the most intuitive and natural.

So, long ago, humans looked at their hands, saw ten fingers, and decided to use a base-10 system. *But how many fingers does a computer have?*

Consider: Computers are built from transistors, and an individual transistor can only be ON or OFF (two options). Similarly, data storage devices can be optical or magnetic. Optical storage devices store data in a specific location by controlling whether light *is* reflected off that location or *is not* reflected off that location (two options). Likewise, magnetic storage devices store data in a specific location by magnetizing the particles in that location with a specific orientation. We can have the north magnetic pole pointing in one direction, or the opposite direction (two options).

Computers can most readily use two symbols, and therefore a base-2 system, or *binary number system*, is most appropriate. The base-10 number system has 10 distinct symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. The base-2 system has exactly two symbols: 0 and 1. The base-10 symbols are termed *digits*. The base-2 symbols are termed binary digits, or *bits* for short. All base-10 numbers are built as strings of digits (such as 6349). All binary numbers are built as strings of bits (such as 1101). Just as we would say that the decimal number 12890 has five digits, we would say that the binary number 11001 is a five-bit number.

The point: All data in a computer is represented in binary. The pictures of your last vacation stored on your hard drive—it’s all bits. The YouTube video of the cat falling off the chair that you saw this morning—bits. Your Facebook page—bits. The tweet you sent—bits. The email from your professor telling you to spend less time on vacation, browsing YouTube, updating your Facebook page and sending tweets—that’s bits too. *Everything is bits.*

To understand how computers work, you have to speak the language. And the language of computers is the binary number system.

The Binary Number System

Consider again the example of a child counting a pile of pennies, but this time in binary. He would begin with the first penny: “1.” The next penny counted makes the total one single group of two pennies. What number is this?

When the base-10 child reached nine (the highest symbol in his scheme), the next penny gave him “one group of ten”, denoted as 10, where the “1” indicated one collection of ten. Similarly, when the base-2 child reaches one (the highest symbol in his scheme), the next penny gives him “one group of two”, denoted as 10, where the “1” indicates one collection of two.

Back to the base-2 child: The next penny makes one group of two pennies and one additional penny: “11.” The next penny added makes two groups of two, which is one group of 4: “100.” The “1” here indicates a collection of two groups of two, just as the “1” in the base-10 number 100 indicates ten groups of ten.

Upon completing the counting task, base-2 child might find that he has one group of four pennies, no groups of two pennies, and one penny left over: 101 pennies. The child counting the same pile of pennies in base-10 would conclude that there were 5 pennies. So, 5 in base-10 is equivalent to 101 in base-2. To avoid confusion when

the base in use if not clear from the context, or when using multiple bases in a single expression, we append a subscript to the number to indicate the base, and write:

5_{10} or 101_2

Just as with decimal notation, we write a binary number as a string of symbols, but now each symbol is a 0 or a 1. To interpret a binary number, we multiply each digit by the power of 2 associated with that digit's position.

For example, consider the binary number 1101 . This number is:

$\begin{array}{ccccccc} & & & \underline{1} & \underline{1} & \underline{0} & \underline{1} \\ & & & \uparrow & \uparrow & \uparrow & \uparrow \\ & & & 2^3 & 2^2 & 2^1 & 2^0 \\ & & & \text{position} & \text{position} & \text{position} & \text{position} \\ & & & \text{(i.e., eight's position)} & \text{(i.e., four's position)} & \text{(i.e., two's position)} & \text{(i.e., one's position)} \end{array}$

Since binary numbers can only contain the two symbols 0 and 1, numbers such as 25 and 1114000 cannot be binary numbers.

We say that all data in a computer is stored in binary—that is, as 1’s and 0’s. It is important to keep in mind that values of 0 and 1 are *logical* values, not the values of a physical quantity, such as a voltage. The actual *physical* binary values used to store data internally within a computer might be, for instance, 5 volts and 0 volts, or perhaps 3.3 volts and 0.3 volts or perhaps *reflection* and *no reflection*. The two values that are used to physically store data can differ within different portions of the same computer. All that really matters is that there are two different symbols, so we will always refer to them as 0 and 1.

A string of eight bits (such as 11000110) is termed a *byte*. A collection of four bits (such as 1011) is smaller than a byte and is hence termed a *nibble*.

Converting Between Binary Numbers and Decimal Numbers

We humans about numbers using the decimal number system, whereas computers use the binary number system. We need to be able to readily shift between the binary and decimal number representations.

Converting a Binary Number to a Decimal Number

To convert a binary number to a decimal number, we simply write the binary number as a sum of powers of 2.

As a shorthand means of converting a binary number to a decimal number, simply write the position value below each bit (i.e., write a “1” below the rightmost bit, then a “2” below the next bit to the left, then a “4” below the next bit to the left, etc.), and then add the position values for those bits that have a value of 1.

For example, to convert the binary number 10101 to decimal, we annotate the position values below the bit values:

$$\begin{array}{ccccc} \underline{1} & \underline{0} & \underline{1} & \underline{0} & \underline{1} \\ 16 & 8 & 4 & 2 & 1 \end{array}$$

Then we add the position values for those positions that have a bit value of 1: $16 + 4 + 1 = 21$. Thus

$$10101_2 = 21_{10}$$

Given a binary number, you can now convert it to the equivalent decimal number. We will now present two different methods for converting in the other direction: from decimal to binary. The first method is more intuitive. The second method is much more readily adaptable to programming on a computer.

Converting a Decimal Number to a Binary Number: Method 1

The first method of converting from a decimal number into a binary number entails expressing the decimal number as a sum of powers of 2. To convert the decimal number x to binary:

Step 1. Find the highest power of two less than or equal to x . The binary representation will have a one in this position. Denote the value of this highest power of 2 as y .

Step 2. Now subtract this power of two (y) from the decimal number (x), denoting the result as z :

$$z = x - y.$$

Step 3. If $z = 0$, you are done. Otherwise, let $x = z$ and return to Step 1 above.

$$2^7 = 128 \quad 2^6 = 64 \quad 2^5 = 32 \quad 2^4 = 16 \quad 2^3 = 8 \quad 2^2 = 4 \quad 2^1 = 2 \quad 2^0 = 1$$

You should “memorize” the binary “nibble – 4-bit” representations of the decimal digits 0 through 15 shown below.

Decimal Number	Binary Number
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Decimal Number	Binary Number
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

You may be wondering about the leading zeros in the table above. For example, the decimal number 5 is represented in the table as the binary number 0101. We could have represented the binary equivalent of 5 as 101, 00101, 00000101, or with any other number of leading zeros. All answers are correct.

Sometimes, though, you will be given the size of a storage location. When you are given the size of the storage location, include the leading zeros to show all bits in the storage location. For example, if told to represent decimal 5 as an 8-bit binary number, your answer should be 0000 0101 – two 4-b nibbles.

Converting a Decimal Number to a Binary Number: Method 2

The second method of converting a decimal number to a binary number entails repeatedly dividing the decimal number by 2, keeping track of the remainder at each step. To convert the decimal number x to binary:

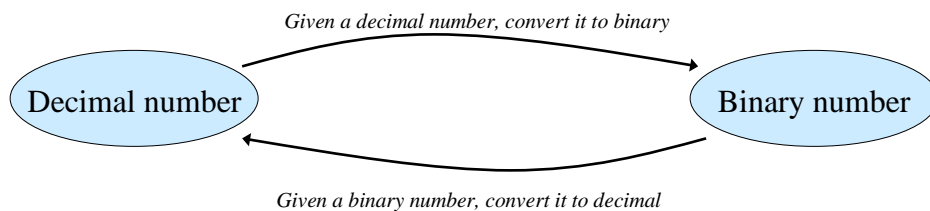
Step 1. Divide x by 2 to obtain a quotient and remainder. The remainder will be 0 or 1.

Step 2. If the quotient is zero, you are finished: Proceed to Step 3. Otherwise, go back to Step 1, assigning x to be the value of the most-recent quotient from Step 1.

Step 3. The sequence of remainders forms the binary representation of the number.

Note that the remainder from the very first division of the decimal number by two provides the least significant bit (i.e., the rightmost bit) in the binary representation. The remainder from each division then provides successive bits, from right to left. The remainder of the final division (which will always be 1) will provide the most significant bit (i.e., the leftmost bit) in the binary representation.

So, now you should be comfortable going back and forth between binary and decimal representations.



Binary Addition

We are familiar with adding decimal (base-10) numbers. But how do we add binary numbers? Consider the equation you encountered in first grade:

$$1 + 1 = 2$$

If you converted each decimal number in this equation to binary, the result would be:

$$1 + 1 = 10$$

We see that when we add two binary ones together, we have a carry into the next column (from the ones column to the twos column). This idea of “carrying over to the next column” when adding binary numbers together is analogous to carrying over when adding decimal numbers.

Think of performing the addition of the decimal numbers 8 and 4:

$$\begin{array}{r} 8 \\ +4 \\ \hline ? \end{array}$$

The result (which we know is the decimal number twelve) cannot be represented by a single decimal symbol (of which the choices are 0, 1, 2, 3, 4, 5, 6, 8 and 9) contained just within the ones column. We have to carry over a quantity of ten to the tens column (where it becomes 1 since it is now in the tens column), and subtract ten from the two numbers we are adding: $8 + 4 - 10 = 2$.

$$\begin{array}{r} 8 \\ +4 \\ \hline 12 \end{array}$$

The same principle carries over to binary addition (or addition in any number system): If the result of the addition is a number too big to be represented as a single digit in the number system, we carry over to the next column, and subtract the quantity from the numbers being added together.

Think of performing the addition of the binary numbers 1 and 1:

$$\begin{array}{r} 1 \\ +1 \\ \hline ? \end{array}$$

The result (which we know to be equivalent to the decimal number two) cannot be represented by a single binary symbol (0 or 1) contained just within the ones column. We have to carry over a quantity of two to the twos column (where it becomes 1 since it is in the twos column), and subtract two from the numbers we are adding: $1 + 1 - 2 = 0$.

$$\begin{array}{r} 1 \\ +1 \\ \hline 10 \end{array}$$

Think of performing the addition of the three binary numbers $1 + 1 + 1$:

$$\begin{array}{r} 1 \\ 1 \\ +1 \\ \hline ? \end{array}$$

The result (which we know to be equivalent to the decimal number three) cannot be represented by a single binary symbol (0 or 1) contained just within the ones column. We have to carry over a quantity of two to the twos column (where it becomes 1 since it is now in the twos column), and subtract two from the three numbers we are adding: $1 + 1 + 1 - 2 = 1$.

$$\begin{array}{r} 1 \\ 1 \\ +1 \\ \hline 11 \end{array}$$

Addition of multi-bit binary numbers is accomplished using the results above, on a column-by-column basis. That is, for each column, use the results:

$$0 + 0 = 0$$

$$1 + 0 = 1$$

$$1 + 1 = 0 \text{ with a carry of 1 to the left}$$

$$1 + 1 + 1 = 1 \text{ with a carry of 1 to the left}$$

For instance, consider the addition of the two binary numbers 11 and 01:

$$\begin{array}{r} 11 \\ + 01 \\ \hline ?? \end{array}$$

Starting with the right column, $1 + 1$ results in 0, with a 1 carried to the left:

$$\begin{array}{r} 1 \\ 11 \\ + 01 \\ \hline ?0 \end{array}$$

Now, in the left column, we are adding $1 + 1 + 0$, which results in 0, with a 1 carried to the left:

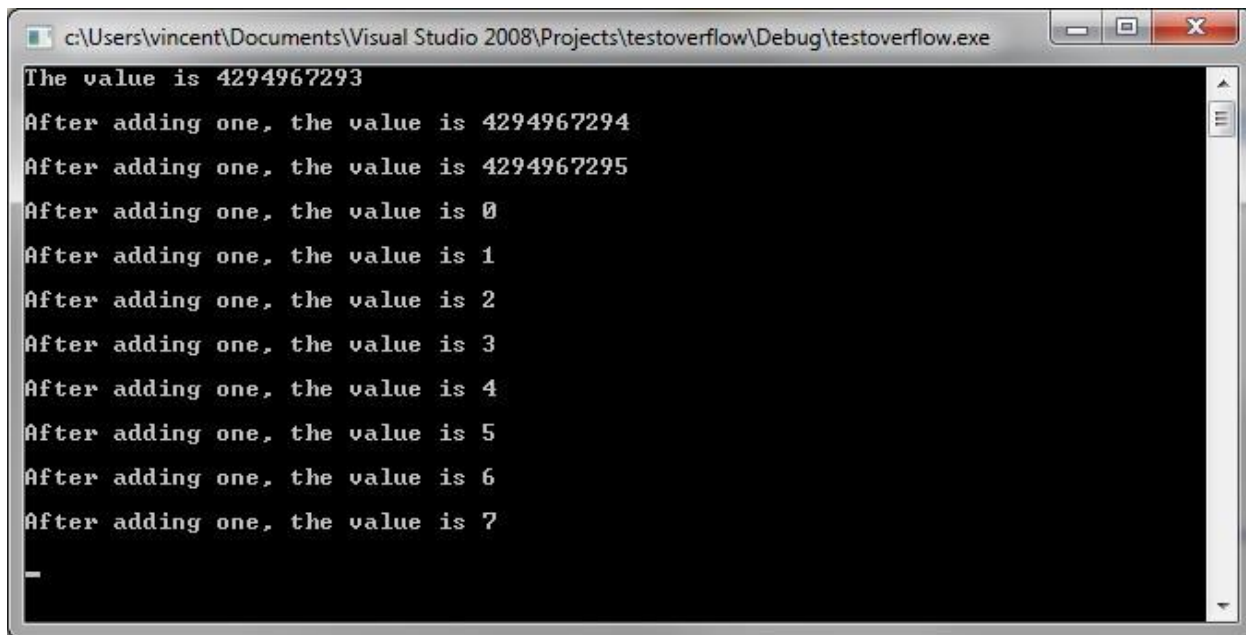
$$\begin{array}{r} 11 \\ 11 \\ + 01 \\ \hline 00 \end{array}$$

Finally, we consider the leftmost column (which consists of just the carryover from the column to its right), resulting in:

$$\begin{array}{r} 11 \\ + 01 \\ \hline 100 \end{array}$$

Overflow

Your friend has written a computer program that initializes a natural number (i.e., a non-negative integer) to the number 4,294,967,293 and then successively adds 1 to the number, printing the result to the screen. He believes that his program must have an error since it produces the results shown below. He asks for your help.



```
c:\Users\vincent\Documents\Visual Studio 2008\Projects\testoverflow\Debug\testoverflow.exe
The value is 4294967293
After adding one, the value is 4294967294
After adding one, the value is 4294967295
After adding one, the value is 0
After adding one, the value is 1
After adding one, the value is 2
After adding one, the value is 3
After adding one, the value is 4
After adding one, the value is 5
After adding one, the value is 6
After adding one, the value is 7
```

Putting aside the obvious question “*Why would your friend have written this program?*”, you decide to help. You notice that the program seems to start out well, adding one and printing out the correct result for the first two iterations, but then, upon reaching the value 4,294,967,295, the addition of one seems to “reset” the value back to zero. The program then seems to resume normal operation, adding one and printing out the correct result.

What’s going on here?

To explore this problem, first consider a computer that uses 4 bits to store integers. The number 15 would be stored as 1111. What would be the result of the calculation $15 + 1$ in binary? Performing binary addition:

$$\begin{array}{r} 1\ 1\ 1\ 1 \\ +\ 0\ 0\ 0\ 1 \\ \hline 1\ 0\ 0\ 0\ 0 \end{array}$$

The result appears to be 10000 which would seem to be correct, since this is, after all, equivalent to 16 in decimal, and that is the right result for $15 + 1$.

Except that this would not be the answer the computer would supply! The correct answer to the binary addition, 10000, requires five bits. But the computer (in our example) stores all integers in four bits. All bits beyond the rightmost four bits are discarded. So, the answer to $15 + 1$ in binary on our 4-bit computer would be zero (i.e., 0000).

his scenario, where an answer requires more bits than the computer has available, is called **overflow**.

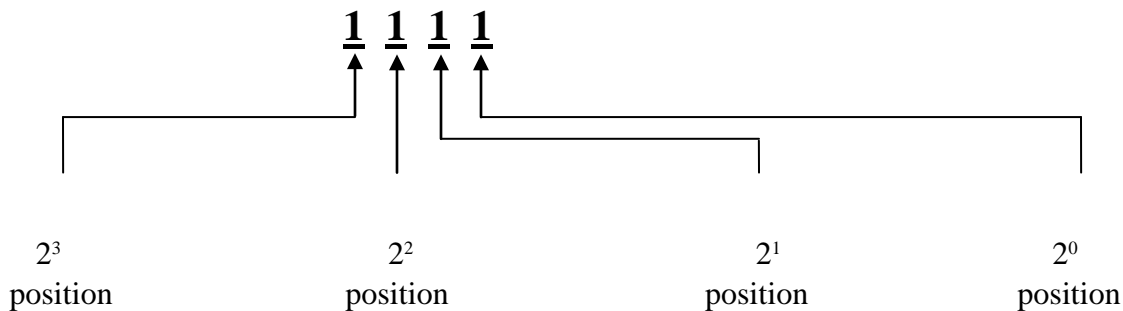
Overflow in the storage of binary numbers is a real practical problem, and has been the bane of many a programmer. The most visually spectacular overflow error occurred in 1996 when the European Space Agency's *Ariane 5* rocket exploded 37 seconds into its maiden launch, sending \$500 million dollars into oblivion. The programmers asked an on-board computer to store a 64-bit velocity in a 16-bit space. Only the rightmost 16-bits were retained. The rocket attempted to respond to the erroneous velocity, swerving itself into disintegration.

So, if a computer stores positive integers in a certain number of bits, what is the largest number that can be stored before overflow occurs?

Using 3 bits the largest binary integer that can be stored is:

Using 4 bits the largest binary integer that can be stored is:

To determine the largest number that can be stored in n bits, first consider the largest number that can be stored in 4 bits:



Note that the most significant bit position is the 2^3 position. If we add 1 to this number (and ignore the fact that overflow will occur), the result will be 10000. Converting this to decimal is easy: There is only a single 1 in the 2^4 position. So, one number larger than 1111 is equal to 2^4 . So 1111 must be equal to $2^4 - 1$.

More generally, the most significant bit position in an n -bit number will be the 2^{n-1} position. The largest number that can be stored in n bits will be a string of n ones (where the leftmost bit is in the 2^{n-1} position). Now, if we add 1 to this string of n ones (and ignore the fact that overflow will occur), the result will be a single 1 in the 2^n position, followed by zeros. So, one number larger than the largest number that can be stored in n bits is equal to 2^n . So, the largest number that can be stored in n bits is $2^n - 1$.

Negative Binary Numbers

We need to represent negative integers also. How do we handle that?

You might be tempted to reply: "Easy, just use a negative sign!" But this won't work. Remember: *everything* must be represented as bits—that is, ones and zeros! A negative sign is neither a one nor a zero.

The most intuitive solution might be the *sign and magnitude* representation. With this notation: Let the leftmost bit represent the sign, say 0 for positive and 1 for negative. The remaining bits represent the magnitude.

For example, suppose we store integers in four bits with the sign and magnitude scheme. Then 3 would be stored as

0011

and -2 would be stored as

1010

As appealing as this scheme might seem at first, it is not used. To see why, consider the addition of 3 and -2. Using addition in the typical way, we would see

$$\begin{array}{r} 3 \\ -2 \\ \hline \end{array} \qquad \begin{array}{r} 0011 \\ 1010 \\ \hline 1101 \end{array}$$

Which is not 1, clearly an error.

So, addition is not straightforward if the signs of the two arguments differ. To make the sign-magnitude scheme work, think about what you would need to do:

You would have to first determine which of the two arguments has the larger magnitude (ignoring the sign bit)

You would then subtract the small magnitude from the larger magnitude

You would then attach to this result the sign of the quantity that had the higher magnitude

While this could be made to work, the resulting hardware would be unnecessarily complex.

As a further annoyance with this scheme, it has two representations for zero. It took mathematicians over a thousand years before they could handle the concept of zero. Surely we should pause before casually introducing a system with two zeros!

The Two's Complement Notation

All computers represent integers in a manner that allows both positive and negative numbers. To represent positive numbers, simply use its binary representation, as we have done before. Negative binary numbers are represented in what is called "*two's complement notation*". To find the two's complement of a binary number, change each 1 to a zero and each zero to a one (i.e., invert all the bits), then add one to this quantity.

What is the two's complement of the binary number $10010010 = ?$

Solution:

$$10010010 \rightarrow 01101101 + 1 = 01101110$$

So...how do we find the representation of a negative decimal number? Do the following: First find the binary representation of the number without the negative sign. Then take the two's complement. The result is the representation of the negative number.

Express the decimal integer -13 as an eight-bit binary number.

Solution: $+13 = 00001101$, $-13 = 11110010 + 1 = 11110011$

There is a neat shortcut to find the two's complement of a binary number. Starting at the right, and moving to the left, leave all bits unchanged until the first 1 is encountered. Leaving this very first 1 unchanged, continue moving to the left inverting all bits from that point onward. For instance, we noted above that the decimal number +53 in binary is

$$0\ 0\ 1\ 1\ 0\ 1\ 0\ 1 = +53$$

To find the two's complement, we start at the right and move to the left, stopping at the first 1 that we encounter, which in this case happens to be the rightmost bit. We leave this bit unchanged, but invert all other bits to the left, resulting in

$$1\ 1\ 0\ 0\ 1\ 0\ 1\ 1 = -53$$

You are likely wondering: Why is the two's complement notation a good representation for negative numbers? Why do we flip all the bits and add one instead of, say, flipping half the bits and adding 5?

Here's why: A binary number, added to its two's complement, results in zero.

For example, let's presume integers are represented using 8 bits. Back in Section 1.3 you determined that the binary representation of 53 was 00110101. Two examples back you determined that -53 was 11001011. Let's add 53 to -53:

$$\begin{array}{r} 53: 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1 \\ -53: 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1 \\ \hline 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \end{array}$$

If we are only using 8 bits, then only the eight least significant bits are retained, and the result is zero (as it should be).

If we represent negative numbers using two's complement notation, then binary addition involving negative numbers will give the correct result. Every computer today uses two's complement notation for the storage of integers.

Two's Complement: A Closer Look

Suppose we have a computer that stores integers in four bits. If we only have non-negative integers, then we can represent the integers 0 – 15 using four bits:

<u>Decimal Number</u>	<u>Binary Number</u>
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Note that we are limited to 16 possible four-bit strings. These 16 unique strings are shown in the right column of the table above. Since we have 16 unique 4-bit strings possible, we can represent 16 unique decimal numbers. These 16 decimal numbers (0 – 15) are shown in the left column of the table above. The assignment of decimal number to bit string is performed by the usual decimal to binary conversion.

Now, suppose we still want to restrict ourselves to storing integers in four bits, but we want to include both positive and negative integers (using two's complement notation, since we know that works!). What integers can we now store, if only four bits are available?

It should be clear that we cannot store the integers -15 to $+15$ within our 4-bit space. The range -15 to $+15$ comprises 31 unique integers, but since we have only 16 possible bit strings available, we can only store 16 unique integers within our four bit allotment.

You should agree that we should store at least the integers -7 to $+7$. But this range will comprise 15 integers, and we have the ability to store sixteen—so there is room for one more! Should it be $+8$? Or should it be -8 ? It can't be both. Let's begin by filling in the integers 0 to 7 = 8 digits and -1 to -8 = 8 digits.

<u>Decimal Number</u>	<u>Binary Number</u>
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111

To summarize: If we use four bits to store integers using two's complement notation, the range of integers that can be stored is -8 through $+7$, and, given the four-bit binary number, we can immediately tell if it is positive or negative by looking at the leftmost bit. If the leftmost bit is a 0, the number is positive. If the leftmost bit is a 1, the number is negative.

How do we “reverse” the two's complement process; i.e., given that a 4-bit two's complement number is stored as, say, 1011, how can I easily determine that the bit string is storing the decimal value -5 ?

Just take the 2's complement again to get its positive representation.

$-5 = 1011$, so take the 2's complement and get $+5 = 0101$

Binary Subtraction

Subtraction in binary entails nothing new! To compute $a - b$, we compute $a + (-b)$ where $-b$ is represented using two's complement notation.

We have overflow if the addition of two positive numbers yields a negative number, or if the addition of two negative numbers yields a positive number. It turns out that there is a quick check that can be done to detect if overflow has occurred: If the carry into the most significant bit (which serves as a sign bit) differs from the carry out from the most significant bit, overflow has occurred.

The Hexadecimal Number System

We often have to deal with large positive binary numbers. Computer programmers must oftentimes look at the contents of a specific item in computer memory. You might, for instance, you may have to look at a variable that is stored at address:

0000000000010010111111101111100

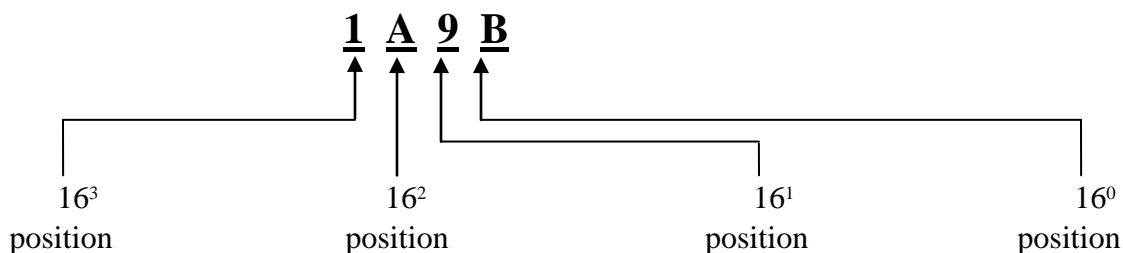
You would probably agree that these long binary strings would be cumbersome to transcribe or to read off to a coworker. Even if you have come to love the binary number system, you would still likely agree that these long strings are too much of a good thing.

Fortunately, large binary numbers can be made much more compact—and hence easier to work with—if represented in base-16, the so-called hexadecimal number system. You may wonder: Binary numbers would also be more compact if represented in base-10—why not just convert them to decimal? The answer, as you will soon see, is that converting between binary and hexadecimal is exceedingly easy—much easier than converting between binary and decimal.

The base-16 hexadecimal number system has 16 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F). Note that the single hexadecimal symbol A is equivalent to the decimal number 10, the single symbol B is equivalent to the decimal number 11, and so forth, with the symbol F being equivalent to the decimal number 15.

Just as with decimal notation or binary notation, we again write a number as a string of symbols, but now each symbol is one of the 16 possible hexadecimal digits (0 through F). To interpret a hexadecimal number, we multiply each digit by the power of **16** associated with that digit's position.

For example, consider the hexadecimal number 1A9B. Indicating the values associated with the positions of the symbols, this number is illustrated as:



Converting a Hexadecimal Number to a Decimal Number

To convert a hexadecimal number to a decimal number, write the hexadecimal number as a sum of powers of 16. For example, considering the hexadecimal number 1A9B above, we convert this to decimal as:

$$1A9B = 1 \times 16^3 + A \times 16^2 + 9 \times 16^1 + B \times 16^0 = 4096 + 10(256) + 9(16) + 11(1) = 6811$$

Converting a Decimal Number to a Hexadecimal Number

The easiest way to convert from decimal to hexadecimal is to use the same division algorithm that you used to convert from decimal to binary, but repeatedly dividing by 16 instead of by 2. As before, we keep track of the remainders, and the sequence of remainders forms the hexadecimal representation.

For example, to convert the decimal number 746 to hexadecimal, we proceed as follows:

	<u>Remainder</u>
16 746	A
16 46	E
16 2	2

So, the decimal number $746 = 2EA$ in hexadecimal

Note that with hexadecimal notation, as with binary and decimal notation, we must be careful that the base is understood. When we speak of the number “23”, do we mean the decimal number 23 (in base 10), or do we mean the hexadecimal number 23 (which happens to equal 35 in base 10)? If the base is not clear from the context, it can be made explicit by including the base as a subscript as in: $23_{16} = 35_{10}$. In ‘C’ we use the prefix 0x to indicate that a number is hexadecimal. That is, instead of writing 23_{16} some texts will use the notation 0x23.

You may be happy to learn that, as a practical matter, programmers and engineers rarely have to convert between decimal and hexadecimal (exams administered by sadistic professors being one of those rare instances!).

Converting a Hexadecimal Number to a Binary Number

Engineers often have to convert between binary and hexadecimal but, as we promised, that is quite simple to do.

We can convert directly from hexadecimal notation to the equivalent binary representation by using the following procedure:

Convert each hexadecimal digit into a four digit binary number, independent of the other hexadecimal digits.

Concatenate the resulting four-bit binary numbers together.

For example, to convert the hexadecimal number 4DA9 to binary, we first convert each hexadecimal digit to a four-bit string:

$$4 = 0100 \quad D = 1101 \quad A = 1010 \quad 9 = 1001$$

Then concatenate the results: The resulting binary number is: 0100 1101 1010 1001. We can drop leading zeros (from the leftmost quartet only!), giving us:

$$4DA9 = 100110110101001$$

Converting a Binary Number to a Hexadecimal Number

Converting from binary to hexadecimal entails reversing the procedure for converting from hexadecimal to binary. Specifically, we can convert directly from binary notation to the equivalent hexadecimal representation by using the following procedure:

Starting at the right, collect the bits in groups of 4

Convert each group of 4 bits into the equivalent hexadecimal digit

Concatenate the resulting hexadecimal digits

For example, to convert 110110101001 to hexadecimal, we collect the bits into groups of 4 starting at the right: 1101 1010 1001, and then we convert each collection of bits into a hexadecimal digit:

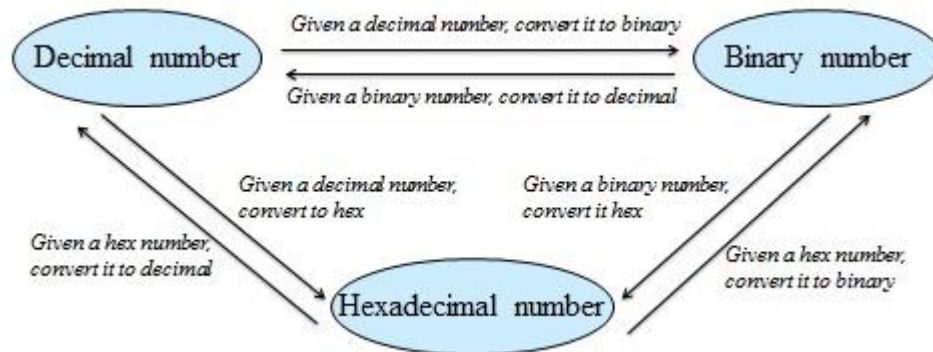
1101	1010	1001
D	A	9

Thus 1101 1010 1001 = DA9

Again, the hexadecimal number system simply provides us with a more convenient means of conveying binary quantities. Instead of saying “The item is at address 1001000100111111”, we can say “The item is at address 913F.”

You may now be comfortable with switching back and forth between binary and hexadecimal, but you may be wondering: *Why is it so easy...why can I get away with converting each hexadecimal digit to binary, independent of the other digits in the number?* After all, we cannot just look at a decimal base-10 number, such as 732, and immediately convert it to base-2 by looking at each of the digits individually.

So, now you should be comfortable going back and forth between binary, decimal and hex representations.



Representation of Characters

You now know how positive and negative integers are represented within the hardware of a computer. We now address the representation of characters, such as letters of the alphabet, punctuation signs and other assorted symbols (e.g., \$, %, ?, etc.).

Characters are stored within the computer using the *American Standard Code for Information Interchange* code—the *ASCII* code—shown in the table on the next page.

The Dec column is base 10, the Hex column is base 16, and the Symbol column is the ASCII character.

Dec	Hex	Symbol	Dec	Hex	Symbol	Dec	Hex	Symbol	Dec	Hex	Symbol
0	0	NUL	16	10	DLE	32	20	(space)	48	30	0
1	1	SOH	17	11	DC1	33	21	!	49	31	1
2	2	STX	18	12	DC2	34	22	"	50	32	2
3	3	ETX	19	13	DC3	35	23	#	51	33	3
4	4	EOT	20	14	DC4	36	24	\$	52	34	4
5	5	ENQ	21	15	NAK	37	25	%	53	35	5
6	6	ACK	22	16	SYN	38	26	&	54	36	6
7	7	BEL	23	17	ETB	39	27	'	55	37	7
8	8	BS	24	18	CAN	40	28	(56	38	8
9	9	TAB	25	19	EM	41	29)	57	39	9
10	A	LF	26	1A	SUB	42	2A	*	58	3A	:
11	B	VT	27	1B	ESC	43	2B	+	59	3B	;
12	C	FF	28	1C	FS	44	2C	,	60	3C	<
13	D	CR	29	1D	GS	45	2D	-	61	3D	=
14	E	SO	30	1E	RS	46	2E	.	62	3E	>
15	F	SI	31	1F	US	47	2F	/	63	3F	?
Dec	Hex	Symbol	Dec	Hex	Symbol	Dec	Hex	Symbol	Dec	Hex	Symbol
64	40	@	80	50	P	96	60	`	112	70	p
65	41	A	81	51	Q	97	61	a	113	71	q
66	42	B	82	52	R	98	62	b	114	72	r
67	43	C	83	53	S	99	63	c	115	73	s
68	44	D	84	54	T	100	64	d	116	74	t
69	45	E	85	55	U	101	65	e	117	75	u
70	46	F	86	56	V	102	66	f	118	76	v
71	47	G	87	57	W	103	67	g	119	77	w
72	48	H	88	58	X	104	68	h	120	78	x
73	49	I	89	59	Y	105	69	i	121	79	y
74	4A	J	90	5A	Z	106	6A	j	122	7A	z
75	4B	K	91	5B	[107	6B	k	123	7B	{
76	4C	L	92	5C	\	108	6C	l	124	7C	
77	4D	M	93	5D]	109	6D	m	125	7D	}
78	4E	N	94	5E	^	110	6E	n	126	7E	~
79	4F	O	95	5F	_	111	6F	o	127	7F	•

Each ASCII symbol is shown with both its hexadecimal representation and its base-10 representation. Suppose we wanted to know how the symbol for the question mark is stored internally within the computer. Scanning the table for the question mark, we note that its hexadecimal value is 3F. Converting this hexadecimal value to binary, we conclude that the question mark is stored as 00111111.

The meaning of each piece of data stored within a computer is defined by the programmer.