Binary Numbers

Bob Brown
Information Technology Department
Southern Polytechnic State University

Positional Number Systems

The idea of "number" is a mathematical abstraction. To use numbers, we must represent them in some way, whether by piles of pebbles or in some other way. It is common to create a **code** for representing numbers. Such codes are called number systems.

Every number system uses symbols to convey information about the value of a number. A **positional** (or **radix**) number system is one in which the value that a symbol contributes to a number is determined by the both symbol itself and the position of the symbol within the number. That's just a fancy way of saying that 300 is far different from 3. Compare the idea of a positional number system with Roman numbers, where X means 10 no matter where in a number it appears.

The decimal number system we use every day is a positional number system. Decimal numbers are also called base 10 or radix 10 numbers. The symbols are the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, the plus and minus signs, and the period or decimal point. The position of each digit within the number tells us the multiplier used with it.

Consider the number 1037. We learned in elementary school to call the rightmost digit the ones place, the next digit the tens place, then the hundreds place, then the thousands place, and so on. Mathematically, 1037 means $1\times10^3 + 0\times10^2 + 3\times10^1 + 7\times10^0$. Each digit in the number is multiplied by some power of ten, starting with 10^0 at the right and increasing by one for each position to the left. The digits of a decimal number are the coefficients of a power series in powers of ten.

Any number to the zeroth power is one, so 10^0 is one and 7×10^0 is just $7\times1 = 7$. A number raised to the first power is itself, so 10^1 is ten, and 3×10^1 is 3×10, or thirty.

In the hundreds place, we have 0×10^2 or "no hundreds." Even though the symbol zero does not contribute to the value of the number, it is important as a placeholder. If we didn't have the zero, we could not distinguish between 1037 and 137.

Using just the digits zero through nine, the decimal number system can express any non-negative integer, however large. The value of an *n*-digit decimal integer is

$$a_{n-1}\times10^{n-1} + a_{n-2}\times10^{n-2} + \ \ldots + a_1\times10^1 \ + a_0\times10^0$$

This can be written more compactly as: $\sum_{i=0}^{n-1} a_i 10^i$

Adding a minus sign to the available symbols lets us express any integer, positive or negative. To express fractions, we use the period symbol. The digit immediately to the right of the period represents 10^{-1} or $^1/_{10}$, the next digit 10^{-2} or $^1/_{100}$, and so on. Although we can represent any integer with decimal numbers, the same is not true of fractions. For example, the fraction $^1/_3$ cannot be exactly represented as a decimal fraction. However, we can make an arbitrarily precise approximation; if 0.33 isn't close enough, we can write 0.333, or even 0.3333333333.

Using Other Bases

The discussion so far has been limited to base 10 or decimal numbers because they are familiar to us. It is possible to represent numbers in positional notation using bases other than 10. An *n*-digit non-negative integer in base *B* would be represented as

$$\sum_{i=0}^{n-1} a_i B^i$$

The only difference between this expression and the one above is that we have substituted some base *B* for 10. The choice of a base isn't entirely arbitrary; we'd like it to be an integer greater than one, and relatively small. Both of these constraints are because a base *B* number system requires *B* symbols. We need at least two symbols so that we can have a zero to serve as a placeholder. We don't want so many symbols that reading and writing numbers becomes unwieldy. In computer science, it is common to deal with numbers expressed as base two, base eight, and base 16.

Binary Numbers

Numbers in base two are called **binary numbers**. A binary number system requires two symbols; we choose 0 and 1. The positions within a binary number have values based on the powers of two, starting with 2^0 in the rightmost position. The digits of a binary number are called *bits*, which is a contraction of *bi*nary digi*ts*.

Consider the binary number 10101. This represents the value $1\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$, or $1\times16 + 0\times8 + 1\times4 + 0\times2 + 1\times1$, or 16 + 4 + 1, or 21. Let's look at the same thing a different way:

1	0	1	0	1	
2^4	2^3	2^2	2^1	2^0	
16	8	4	2	1	
16		+ 4		+ 1	= 21

The first row is the binary number we want to examine. On the second row starting at the right, we write the power of two that corresponds to each position. The rightmost position is 2^0 and the power increases by one with each position to the left.

The third row is the decimal value of each of the powers of two. Notice that each of the numbers is twice the value of its predecessor. You simply start with one in the rightmost position and double each time you move left.

The decimal value of each digit is the digit itself, zero or one, multiplied by the power of two indicated by the digit's position. If the digit is a one, we copy the power of two to the fourth row; if the digit is a zero, we do not. This is equivalent to multiplying each positional value by the associated binary digit. Finally, we add across the bottom row to get the decimal value of the binary number.

As students of computer science, you will find it convenient to memorize the values of the first several powers of two, plus certain other values like 2^{10}, 2^{16}, and 2^{20}. You can easily find the value of any small power of two by starting with one you know and doubling until you

reach the desired value. For example, if you know 2^{16}, you can find 2^{18} by doubling twice. If necessary, you can start with $2^0 = 1$ and double until you reach the value you need.

Usually the base of a number will be clear from the context; if necessary, the base is indicated by a subscript following the number, so we could write, for example, $1010_2 = 10_{10}$.

Why Binary?

We've taken some trouble to describe a number system based only on zero and one. It is appropriate to digress briefly to explain why we choose to use binary numbers instead of the familiar decimal system when building computers. The answer is *reliability*. It turns out to be easy to design electronic circuits that can distinguish between on and off, or between positive and negative. It is much harder to build circuits that can reliably discriminate among several states. Seemingly identical electronic components will be slightly different even when new because of manufacturing tolerances. These differences are magnified with age and with differences in operating environment.

Consider a decimal computing machine in which we choose to represent the digits zero through nine with signals of zero through nine volts. We design the machine so that an actual signal of 6.8 volts is interpreted as the digit seven, allowing some tolerance for error. We also decide that 6.4 volts represents six. What do we do with a voltage of 6.5? Does this represent seven or six? With this scheme, a difference of 0.5 volts, or five percent, causes an error that cannot be detected.

With binary numbers, we need only the symbols zero and one. If we say that zero volts represents a zero and nine volts represents a one, we can interpret anything less than 4.5 volts as zero, anything greater as one. This design can tolerate an error of nearly 50% and still produce correct results.

How High Can We Count?

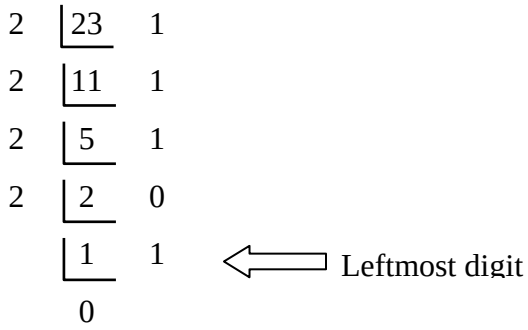
With pencil and paper, we can write down any number we can imagine using either the decimal or binary number systems. If we need more digits, we just write them. With computing machinery, the number of bits available for a number is likely to be fixed by the architecture. There may be ways of representing larger numbers, but these are likely to be painful. So, the question, "How high can we count given a fixed number of bits?" becomes an important one. Fortunately, it's easy to answer.

An *n*-bit binary number has 2^n possible values. This is easy to see. A single bit has two possible values, zero and one. With two bits, you get four values: 00, 01, 10, and 11. Three bits can generate eight distinct combinations, and so on. Of the 2^n possible values of an *n*-bit number, one will be all zeroes and represent the value zero. So the largest value that can be represented using an *n*-bit number is $2^n - 1$. An eight bit binary number has 2^8 (256) possible values, but since one of those values represents zero, the largest possible number is $2^8 - 1$ or 255.

There's another implication to the fact that in computers, binary numbers are stored in fixed-size "words." It is that each binary number will be the same number of bits. For unsigned integers, this is accomplished by padding on the left with zeroes.

Converting Decimal to Binary

We can convert decimal to unsigned binary almost as easily as we converted binary to decimal. The process is to divide the decimal number repeatedly by two and keep track of the remainders. Let's convert 23_{10} to binary.



We divide 23 by 2 and get 11 with remainder one. Write the remainder to the side and the quotient below. Divide 11 by two to get a quotient of five and a remainder of one. Divide five by two to get a quotient of two and a remainder of one. Divide two by two to get a quotient of one and a remainder of zero. Finally, divide one by two to get a quotient of zero and a remainder of one. The algorithm terminates when the quotient reaches zero.

The remainders represent the bits of the binary number, with the last remainder being the leftmost digit. Copying from bottom to top, we get 10111, and $23_{10} = 10111_2$.

The first division by two tells us how many twos there are in 23; there are 11. The remainder tells us how many ones were left over. That number will always be zero or one. In this case, it is one. The second division tells how many fours there are in 23, and there are five. The remainder tells how many twos were left, in this case one. Successive divisions give the number of eights and sixteens. If the number being converted were larger, we would keep going to find 32s, 64s, and so on.

Hexadecimal Numbers

Binary numbers are essential for the computers, but binary numbers more than a few digits long are difficult to transcribe accurately. The hexadecimal (base 16) and octal (base 8) number systems can be used as number systems in their own right, but in computer science they are most often used as a shorthand for binary numbers. Since the bases of both systems are powers of two, translating between either base and binary is easy.

Like decimal and binary numbers, the *hexadecimal*, or base 16 number system is a positional number system. We know that there must be 16 symbols, and we choose 0, 1, …, 9, A, B, C, D, E, and F. Symbols 0 through 9 have the same unit values they have in the decimal system, but of course the positional multiplier is different. Hexadecimal (or *hex*) **A** has the value 10_{10}, **B** is 11_{10}, **C** is 12_{10}, **D** is 13_{10}, **E** is 14_{10}, and **F** is 15_{10}.

The positions in a hexadecimal number have as their values powers of 16, starting with 16^0 at the right, then 16^1, 16^2 or 256, 16^3 or 4096, and so on. Four hexadecimal digits let us represent numbers up to $15\times16^3 + 15\times16^2 + 15\times16^1 + 15$, or $15\times4096 + 15\times256 + 15\times16 + 15$, or 61,440 + 3840 + 240 + 15, or 65,535. This number would be represented as FFFF. A value of 0100_{16} is equal to 256_{10}.

Hexadecimal numbers can be used as a kind of shorthand for binary numbers, to avoid writing out long strings of ones and zeroes. Study the following table:

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

Binary	*Hex*	*Decimal*
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

As you can see, each hex digit is exactly equivalent to one of the possible combinations of four binary digits, so we could write 7_{16} instead of 0111_2. This works for numbers larger than four bits or one hex digit. $7A_{16}$ is equivalent to 01111010_2. Four hex digits let us express a 16-bit binary number in four symbols instead of 16.

It is common to use indications other than a subscript 16 to identify numbers as hexadecimal when it is not clear from the context. The following are all examples of indicators of hexadecimal numbers: x'7A', 0x7A, and 7Ax. In the Motorola 68000 assembler we will be using in CS2224, hexadecimal numbers are indicated by a dollar sign, so \$08 is 8_{16}.

Converting from hexadecimal to binary is easy: for each hex digit, write the four binary digits which have the same value. For example, to convert $4C_{16}$ to binary, we first write 0100 which is the binary equivalent of 4_{16}, then we write 1100 which is the binary equivalent of C_{16}, so $4C_{16} = 01001100_2$.

To convert a binary number to hexadecimal, start at the right and divide the number into groups of four bits. If the last group is fewer than four bits, supply zeroes on the left. (If the binary number contains a radix point, move from right to left on the integer part and from left to right on the fraction part.) Then, for each group of four bits, write the hex digit which has the same value.

For example, we will convert 1100011101 to hex.

```
0011 0001 1101
 3    1    D
```

We first divide the binary number into groups of four bits, working from the right. The last (leftmost) group had only two digits, so two zeroes were supplied on the left. The leftmost group of bits has the numeric value three, and we write a three as the hex digit for that group.

The next group has the numeric value one. The rightmost group has the numeric value 13, or hex D. We have converted 1100011101_2 to $31D_{16}$.

Octal Numbers

The *octal* number system is a positional number system with base eight. Like hexadecimal numbers, octal numbers are most frequently used as shorthand for binary numbers. Octal numbers are seen less often than hexadecimal numbers, and are commonly associated with Unix-like operating systems. The octal digits are 0, 1, 2, 3, 4, 5, 6, and 7. These digits have the same unit values as the corresponding decimal digits. Each octal digit encodes three binary digits as shown in the table below.

Binary	*Octal*	*Decimal*
000	0	0
001	1	1
010	2	2
011	3	3
100	4	4
101	5	5
110	6	6
111	7	7

Octal numbers are sometimes indicated by a leading zero.

Numbers are converted from octal to binary one digit at a time. For each octal digit, write down the three binary digits which represent the same value. To convert 124_8 to binary we write 001 010 100, and $124_8 = 001010100_2$. Sometimes octal numbers are used to represent eight-bit quantities. Representing eight bits requires three octal digits, which translate to nine bits. In this case, the leftmost bit, which should be a zero, is discarded.

Converting a binary number to octal follows the same process as converting a hexadecimal number except that the binary number is divided into groups of three bits. To convert 01001100_2 to octal, we divide the number into groups of three bits, starting from the right, then write down the corresponding octal digit for each group.

001 001 100

1 1 4

In this example, it was necessary to supply an extra zero on the left to make three bits. We have converted 01001100_2 to 114_8.

To convert octal numbers to hexadecimal, or vice-versa, first convert to binary, then convert to the desired base by grouping the bits.

Binary Addition

Since we've talked about binary numbers as the basis for the electronic circuits for computers, it won't surprise you that we can do arithmetic on binary numbers. All the operations of ordinary arithmetic are defined for binary numbers, and they work much the same as you are used to. Let's look at the rules for binary addition:

				1
0	0	1	1	+1
+0	+1	+0	+1	+1
0	1	1	10	11

The first three of those don't require any explanation, but the fourth and fifth might. Look at the fourth rule and recall that the result is a binary number, so 10_2 represents one two and no ones, and in binary one plus one is two, exactly as you would expect. The rule says that 1+1=0, with one to *carry* to the next place. This is the same principle as carrying numbers in decimal addition, except that we carry when the partial sum is greater than one. The fifth rule adds three ones to get a one as the partial sum and another one to carry. We've written 1+1+1=3, which is what we expect.

Now we will add two binary numbers with more than one bit each so you can see how the carries "ripple" left, just as they do in decimal addition.

	1	1	1		
	0	0	1	1	0
+	0	1	1	1	1
	1	0	1	0	1

The three carries are shown on the top row. Normally, you would write these down as you complete the partial sum for each column. Adding the rightmost column produces a one with no carry; adding the next column produces a zero with one to carry. Work your way through the entire example from right to left. Then convert the addend, augend, and sum to decimal to verify that we got the right answer.

One can also express the rules of binary addition with a *truth table.* This is important because there are techniques for designing electronic circuits, which compute functions expressed by truth tables. The fact that we can express the rules of binary addition as a truth table implies that we can design a circuit which will perform addition on binary numbers, and that turns out to be true.

We only need to write the rules for one column of bits; we start at the right and apply the rules to each column in succession until the final sum is formed. Call the bits of the addend and augend **A** and **B**, and the carry in from the previous column $\mathbf{C}_i$. Call the sum **S** and the carry out $\mathbf{C}_o$. The truth table for one-bit binary addition looks like this:

A	B	C_i	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

This says if all three input bits are zero, both **S** and $\mathbf{C_o}$ will be zero. If any one of the bits is one and the other two are zero, **S** will be one and $\mathbf{C_o}$ will be zero. If two bits are ones, **S** will be zero and $\mathbf{C_i}$ will be one. Only if all three bits are ones will both **S** and $\mathbf{C_o}$ be ones.

That's all there is to binary addition. It's remarkably similar to decimal addition. As you would expect, the other arithmetic operations are also defined for binary numbers.

Negative Numbers

So far we have dealt only with non-negative integers — whole numbers zero or greater. For a computer to be useful, we must be able to handle binary negative numbers and fractions. With decimal numbers, we use the minus sign to indicate a negative number. In a computer, we have only the symbols zero and one, so we have to use zero and one to represent positive and negative. It turns out to be convenient to use zero to indicate positive and one to indicate negative.

We could just tack a "sign bit" onto the left of a binary number and let it indicate positive or negative. In fact, this can be made to work; the concept is called *signed magnitude*. There's a problem with signed magnitude: it has two representations for zero. Consider an eight-bit word: 00000000 is "plus zero" and 10000000 is "minus zero." Since testing for zero is something that's done very frequently in computer programming, we would like to develop a better idea.

The better idea is something called *two's complement*. The sign still resides in the leftmost bit, and positive numbers are treated just like the unsigned integers we are already used to except that results are never allowed to flow over into the sign bit. For negative numbers, the sign bit represents the *negative of the power of two* corresponding to its bit position. An example will make this clearer. Consider 10000000, an eight-bit binary number. The leftmost bit, or sign bit, corresponds to 2^7 or 128. We are representing a negative number, so we consider the sign bit to have the value –128. The remaining bits are treated as positive values that are added back to form the final value. So, 10000000 is –128; it is -2^7 with nothing added back. 10000001 is –127, or –128 + 1, 10000010 is –126, and so on. 11111111 represents –1, –128 with 127 added back. Zero is 00000000, and there's no confusion between positive and negative zeroes.

The two's complement representation has its own drawback. Notice that in eight bits we can represent –128 by writing 10000000. The largest positive number we can represent is 01111111 or +127. Two's complement is *asymmetric about zero*. For any size binary number, there is one more negative number than there are positive numbers. This is because, for any binary number, the number of possible bit combinations is even. We use one of those combinations for zero, leaving an odd number to be split between positive and negative. Since we want zero to be represented by all binary zeros, there's no way to escape from having one more negative number than positive.

If you think of a two's complement number as a large negative number with positive numbers added back, you could conclude that it would be difficult to form a two's complement. It turns out that there's a method of forming the two's complement that is very easy to do with either a pencil or a computer:

- Take the complement of each bit in the number to be negated. That is, if a bit is a zero, make it a one, and vice-versa.
- To the result of the first step, add one as though doing unsigned arithmetic.

Let's do an example: we will find the two's complement representation of –87. We start with the binary value for 87, or 01010111. Here are the steps:

01010111	original number
10101000	each bit complemented, or "flipped"
+ 1	add 1 to 10101000
10101001	this is the two's complement, or –87.

We can check this out. The leftmost bit represents –128, and the remaining bits have positive values which are added back. We have –128 + 32 + 8 + 1, or –128 + 41 = –87. There's another way to check this. If you add equivalent negative and positive numbers, the result is zero, so –87 + 87 = 0. Does 01010111 + 10101001 = 0? Perform the addition and see.

There's another way to find the two's complement of a binary number: Copy bits *from the right* until you have copied a one bit, then invert the remaining bits. Use whichever method seems easiest for you.

Fractions

In ordinary decimal numbers, we represent fractions as negative powers of ten, and we mark the division between the integer and fraction parts with a "decimal point." The same principle applies to binary numbers. 0.1_2 is 2^{-1} or $^1/_2$. 0.11_2 is $2^{-1} + 2^{-2}$ or $^1/_2 + ^1/_4 = ^3/_4$. As with decimal fractions, not all fractional values can be represented exactly, but we can get arbitrarily close by using more fraction bits.

Unhappily, we don't have a symbol we can use for the "binary point." A programmer who wants to use binary fractions must pick a location for the implied binary point and scale all numbers to maintain binary point alignment.

Exercises

1. Convert the following unsigned binary numbers to decimal:

   10101
   10000
   01111
   11111

2. Convert the following decimal numbers to binary:

   32
   127
   255
   275

3. Which of the following binary numbers are even? How can you tell by inspection whether a binary number is even or odd?

   101010
   101011
   111111
   111110

4. Convert 42_{10} to binary, negate it by forming the two's complement, and compute –42 + 42 using binary numbers. What answer do you expect and why?

5. Write the binary equivalent of $3^{1}/_{16}$. To make your work clear, use a period as a "binary point."

6. Using only the methods and material presented above, suggest a strategy for performing binary subtraction.

7. Which of the following are valid hexadecimal numbers?
   BAD DEAD CABBAGE ABBA 127

8. Convert the following binary numbers to hexadecimal.

   00000000
   10001000
   00011010
   11111111

9. Convert the binary numbers in Exercise 8 to octal.

10. Convert the following octal numbers to both binary and hexadecimal.

    377
    127
    4066
    01

11. Explain why there is one more negative number than there are positive numbers for a given number of bits in two's complement representation.