Chapter 0 - Numeral Systems and Storing Data in Memory

0 Numeral Systems and Storing Data in Memory

This chapter describes numeral systems and the way in which computer hardware represents and manages information.

0.1 Number and Computing

Numbers are crucial to computing. In addition to using a computer to execute numeric computations, all types of information that we store and manage using a computer are ultimately stored as numbers. At the lowest level, computers store all information using binary digits 0 and 1, or *bits*. So to begin our exploration of computers, we need to first begin by exploring numbers.

First, let's recall that numbers can be classified into all sorts of categories. There are natural numbers, negative numbers, rational numbers, irrational numbers, and many others that are important in mathematics but not to the understanding of computing. Let's review the relevant category definitions briefly.

First, let's define the general concept of a number: A **number** is a unit belonging to an abstract mathematical system and is subject to specified laws of succession, addition, and multiplication. That is, a number is a representation of a value, and certain arithmetic operations can be consistently applied to such values.

> **Number** A unit of an abstract mathematical system subject to the laws of arithmetic

Now let's separate numbers into categories. A **natural number** is the number 0 or any number obtained by repeatedly adding 1 to this number. Natural numbers are the ones we use in counting. A **negative number** is less than zero and is opposite in sign to a positive number. An **integer** is any of the natural numbers or any of the negatives of these numbers. A **rational number** is an integer or the quotient of two integers—that is, any value that can be expressed as a fraction.

> **Natural number** The number 0 and any number obtained by repeatedly adding 1 to it
>
> **Negative number** A value less than 0, with a sign opposite to its positive counterpart
>
> **Integer** A natural number, a negative of a natural number, or zero
>
> **Rational number** An integer or the quotient of two integers (division by zero excluded)

Some of the material in this chapter may already be familiar to you. Certainly some of the underlying ideas should be. You probably take for granted some basic principles of numbers and arithmetic because you've become so used to them. Part of our goal in this chapter is to remind you of those underlying principles and to show you that they apply to all number systems. Then the idea that a computer uses binary values—that is, 1s and 0s—to represent information should be less mysterious.

0.2 Positional Notation

How many ones are there in 943? That is, how many actual things does the number 943 represent? Well, in grade school terms, you might say there are 9 hundreds plus 4 tens plus 3 ones. Or, said another way, there are 900 ones plus 40 ones plus 3 ones. So how many ones are there in 754? 700 ones plus 50 ones plus 4 ones. Right? Well, maybe. The answer depends on the *base* of the number system you are using. This answer is correct in the base-10, or decimal, number system, which is the number system humans use every day. But that answer is not correct in other number systems.

The **base** of a number system specifies the number of digits used in the system. The digits always begin with 0 and continue through one less than the base. For example, there are 2 digits in base-2: 0 and 1. There are 8 digits in base-8: 0 through 7. There are 10 digits in base-10: 0 through 9. The base also determines what the positions of digits mean. When you add 1 to the last digit in the number system, you have a carry to the digit position to the left.

> **Base** The foundational value of a number system, which dictates the number of digits and the value of digit positions

Numbers are written using **positional notation**. The rightmost digit represents its value multiplied by the base to the zeroth power. The digit to the left of that one represents its value multiplied by the base to the first power. The next digit represents its value multiplied by the base to the second power. The next digit represents its value multiplied by the base to the third power, and so on. You are so familiar with positional notation that you probably don't think about it. We used it instinctively to calculate the number of ones in 943.

> **Positional notation** A system of expressing numbers in which the digits are arranged in succession, the position of each digit has a place value, and the number is equal to the sum of the products of each digit by its place value

$$\begin{aligned} 9 * 10^2 = 9 * 100 &= 900 \\ + 4 * 10^1 = 4 * \ 10 &= \ 40 \\ + 3 * 10^0 = 3 * \ \ 1 &= \ \ 3 \\ &\overline{\quad 943} \end{aligned}$$

Fig 0.1 Sum using base-10 positional notation

A more formal way of defining positional notation is to say that the value is represented as a polynomial in the base of the number system. But what is a polynomial? A polynomial is a sum of two or more algebraic terms, each of which consists of a constant multiplied by one or more variables raised to a nonnegative integral power. When defining positional notation, the variable is the base of the number system. Thus 943 is represented as a polynomial as follows, with x acting as the base:

$$9 * x^2 + 4 * x^1 + 3 * x^0$$

Let's express this idea formally. If a number in the base-R number system has n digits, it is represented as follows, where d_i represents the digit in the i^{th} position in the number:

$$d_n * R^{n-1} + d_{n-1} * R^{n-2} + \cdots + d_2 * R^1 + d_1 * R^0$$

and since $R^1 = R$ and $R^0 = 1$, it can simply be re-written as:

$$d_n * R^{n-1} + d_{n-1} * R^{n-2} + \cdots + d_2 * R + d_1$$

Look complicated? Let's look at a concrete example: 63578 in base-10. Here n is 5 (i.e. the number has five digits), and R is 10 (i.e. the base). The formula says that the fifth digit (last digit on the left) is multiplied by the base to the fourth power; the fourth digit is multiplied by the base to the third power; the third digit is multiplied by the base to the second power; the second digit is multiplied by the base to the first power; and the first digit is not multiplied by anything:

$$6 * 10^4 + 3 * 10^3 + 5 * 10^2 + 7 * 10^1 + 8$$

In the previous calculation, we assumed that the number base is 10. This is a logical assumption because our number system *is* base-10. However, there is no reason why the number 943 couldn't represent a value in base-13. If so, to determine the number of ones, we would have to convert it to base-10.

$$\begin{aligned} 9 * 13^2 &= 9 * 169 = 1521 \\ +\,4 * 13^1 &= 4 * 13 = 52 \\ +\,3 * 13^0 &= 3 * 1 = 3 \\ & \qquad\qquad\quad 1576 \end{aligned}$$

Fig 0.2. Sum using base-13 positional notation

Therefore, 943 in base-13 is equal to 1576 in base-10. Keep in mind that these two numbers have an equivalent value. That is, both represent the same number of things. If one bag contains 1576 (base-10) beans and a second bag contains 943 (base-13) beans, then both bags contain the exact same number of beans. Number systems just allow us to represent values in various ways.

Note that in base-10, the rightmost digit is the "ones" position. In base-13, the rightmost digit is also the "ones" position. In fact, this is true for any base, because anything raised to the power 0 is 1.

Why would anyone want to represent values in base-13? It isn't done very often, granted, but it is sometimes helpful to understand how it works. For example, a computing technique called *hashing* takes numbers and scrambles them, and one way to scramble numbers is to interpret them in a different base.

Other bases, such as base-2 (binary), are particularly important in computer processing. Let's explore these bases in more detail.

0.3 Binary, Octal and Hexadecimal

The base-2 (binary) number system is particularly important in computing. It is also helpful to be familiar with number systems that are powers of 2, such as base-8 (octal) and base-16 (hexadecimal). Recall that the base value specifies the number of digits in the number system. Base-10 has ten digits (0–9), base-2 has two digits (0–1), and base-8 has eight digits (0–7). Therefore, the number 943 could not represent a value in any base less than base 10, because the digit 9 doesn't exist in those bases. It is, however, a valid number in base-10 or any base higher than that. Likewise, the number 2074 is a valid number in base-8 or higher, but it simply does not exist (because it uses the digit 7) in any base lower than that.

What are the digits in bases higher than 10? We need symbols to represent the digits that correspond to the decimal values 10 and beyond. In bases higher than 10, we use letters as digits. We use the letter A to represent the number 10, B to represent 11, C to represent 12, and so forth. Therefore, the 16 digits in base-16 are:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}$$

Let's look at values in octal, hexadecimal, and binary to see what they represent in base-10. For example, let's calculate the decimal equivalent of 754 in octal (base-8). As before, we just expand the number in its polynomial form and add up the numbers.

$$\begin{aligned} 7 * 8^2 &= 7 * 64 = 448 \\ +5 * 8^1 &= 5 * 8 = 40 \\ +4 * 8^0 &= 4 * 1 = 4 \\ & \qquad\qquad 492 \end{aligned}$$

Fig 0.3. Sum using base-8 positional notation (converting octal number 754 to decimal)

Let's convert the hexadecimal number ABC to decimal:

$$\begin{aligned} \mathrm{A} * 16^2 &= 10 * 256 = 2560 \\ +\mathrm{B} * 16^1 &= 11 * 16 = 176 \\ +\mathrm{C} * 16^0 &= 12 * 1 = 12 \\ & \qquad\qquad 2748 \end{aligned}$$

Fig 0.4. Sum using base-16 positional notation (converting hexadecimal number ABC to decimal)

Note that we perform the exact same calculation to convert the number to base-10. We just use a base value of 16 this time, and we have to remember what the letter digits represent. After a little practice you won't find the use of letters as digits that strange.

Finally, let's convert a binary (base-2) number 1010110 to decimal. Once again, we perform the same steps—only the base value changes:

$$
\begin{array}{rcrcr}
1 * 2^6 & = & 1 * 64 & = & 64 \\
+0 * 2^5 & = & 0 * 32 & = & 0 \\
+1 * 2^4 & = & 1 * 16 & = & 16 \\
+0 * 2^3 & = & 0 * 8 & = & 0 \\
+1 * 2^2 & = & 1 * 4 & = & 4 \\
+1 * 2^1 & = & 1 * 2 & = & 2 \\
+0 * 2^0 & = & 0 * 1 & = & \underline{0} \\
 & & & & 86
\end{array}
$$

Fig 0.5. Sum using base-2 positional notation (converting binary number 1010110 to decimal)

Recall that the digits in any number system go up to one less than the base value. To represent the base value in any base, you need two digits. A 0 in the rightmost position and a 1 in the second position represent the value of the base itself. Thus 10 is ten in base-10, 10 is eight in base-8, and 10 is sixteen in base-16. Think about it. The consistency of number systems is actually quite elegant.

Addition and subtraction of numbers in other bases are performed exactly like they are on decimal numbers.

0.4 Arithmetic in Other Bases

Recall the basic idea of arithmetic in decimal: $0 + 1$ is 1, $1 + 1$ is 2, $2 + 1$ is 3, and so on. Things get interesting when you try to add two numbers whose sum is equal to or larger than the base value—for example, $1 + 9$. Because there isn't a symbol for ten, we reuse the same digits and rely on position. The rightmost digit reverts to 0, and there is a carry into the next position to the left. Thus $1 + 9$ equals 10 in base-10.

The rules of binary arithmetic are analogous, but we run out of digits much sooner. That is, $0 + 1$ is 1, and $1 + 1$ is 0 with a carry. Then the same rule is applied to every column in a larger number, and the process continues until we have no more digits to add. The example below adds the binary values 101110 and 11011. The carry value is marked above each column in color.

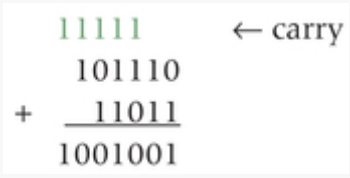


Fig 0.6. Addition of binary numbers 101110 and 11011.

We can convince ourselves that this answer is correct by converting both operands to base-10, adding them, and comparing the result: 101110 is 46, 11011 is 27, and the sum is 73. Of course, 1001001 is 73 in base-10.

The subtraction facts that you learned in grade school were that $9 - 1$ is 8, $8 - 1$ is 7, and so on, until you try to subtract a larger digit from a smaller one, such as $0 - 1$. To accomplish this feat, you have to "borrow one" from the next left digit of the number from which you are subtracting. More precisely, you borrow one power of the base. So, in base-10, when you borrow, you borrow 10. The same logic applies to binary subtraction. Every time you borrow in a binary subtraction, you borrow 2. Here are two examples with the borrowed values marked above.

Fig 0.7. Subtraction of binary numbers 110 from 111001 and 111101.

Once again, you can check the calculation by converting all values to base 10 and subtracting to see if the answers correspond.

0.5 Power-of-2 Number Systems

Binary and octal numbers share a very special relationship: Given a number in binary, you can read it off in octal; given a number in octal, you can read it off in binary. For example, take the octal number 754. If you replace each digit with the binary representation of that digit, you have 754 in binary. That is, 7 in octal is 111 in binary, 5 in octal is 101 in binary, and 4 in octal is 100 in binary, so 754 in octal is $111\ 101\ 100$ in binary.

To facilitate this type of conversion, the table below shows counting in binary from 0 through 10 with their octal and decimal equivalents.

Binary	Octal	Decimal	Hexadecimal
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F
10000	20	16	10

To convert from binary to octal, you start at the rightmost binary digit and mark the digits in groups of threes. Then you convert each group of three to its octal value:

$$\begin{array}{ccc} 111 & 101 & 100 \\ 7 & 5 & 4 \end{array}$$

Fig 0.8. Converting binary number 111101100 to octal

Let's convert the binary number 1010110 to octal, and then convert that octal value to decimal. The answer should be the equivalent of 1010110 in decimal, or 86.

$$\begin{array}{cccc} \underline{1} & \underline{010} & \underline{110} & \\ 1 & 2 & 6 & \\ & & & \\ 1 * 8^2 & = 1 * 64 & = & 64 \\ + 2 * 8^1 & = 2 * 8 & = & 16 \\ + 6 * 8^0 & = 6 * 1 & = & \underline{6} \\ & & & 86 \end{array}$$

Fig 0.9. Converting of binary number 1010110 to decimal

The reason that binary can be immediately converted to octal and octal to binary is that 8 is a power of 2. There is a similar relationship between $binary$ and $hexadecimal$. Every hexadecimal digit can be represented in four binary digits. Let's take the binary number 1010110 and convert it to hexadecimal by marking the digits from right to left in groups of four.

$$\begin{array}{cccc} \underline{101} & \underline{0110} & & \\ 5 & 6 & & \\ & & & \\ 5 * 16^1 & = 5 * 16 & = & 80 \\ + 6 * 16^0 & = 6 * 1 & = & \underline{6} \\ & & & 86 \end{array}$$

Fig 0.10. Converting of binary number 1010110 to hexadecimal

Now let's convert ABC in hexadecimal to binary. It takes four binary digits to represent each hex digit. A in hexadecimal is 10 in decimal and therefore is 1010 in binary. Likewise, B in hexadecimal is 1011 in binary, and C in hexadecimal is 1100 in binary. Therefore, ABC in hexadecimal is 101010111100 in binary.

Rather than confirming that 101010111100 is 2748 in decimal directly, let's mark it off in octal and convert the octal.

$$\begin{array}{cccc} \underline{101} & \underline{010} & \underline{111} & \underline{100} \\ 5 & 2 & 7 & 4 \end{array}$$

Fig 0.11. Converting of binary number 101010111100 to octal before using table

Then, converting octal to decimal using:

$5 * 8^3 + 2 * 8^2 + 7 * 8^1 + 4 * 8^0 = 2560 + 128 + 56 + 4 = 2748$

Thus, 5274 in octal is 2748 in decimal.

In the next section, we show how to convert base-10 numbers to the equivalent number in another base.

0.6 Converting from Base-10 to Other Bases

Converting base-10 numbers involves dividing by the base into which you are converting the number. The division produces a quotient and a remainder. The remainder becomes the next digit in the new number (going from right to left), and the quotient replaces the number to be converted. The process continues until the quotient is zero. This algorithm can be written as:

> **while** (the quotient is not zero)
> Divide the decimal number by the new base
> Make the remainder the next digit to the left in the answer
> Replace the decimal number with the quotient
> **endwhile**

The `while` statement tells us to repeat the next three lines until the quotient becomes zero. Let's convert decimal 2748 to hexadecimal. As we've seen in previous examples, the answer should be ABC.

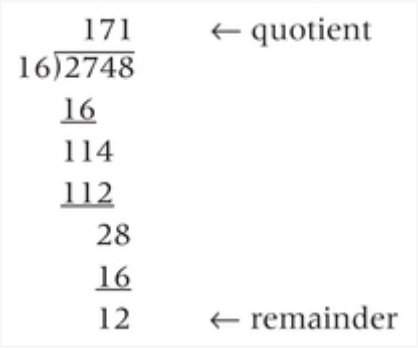


Fig 0.1.12a. Long division of $2748 \div 16$

The remainder (12) is the first digit in the hexadecimal answer, represented by the digit C. So the answer so far is C. Since the quotient is not zero, we divide it (171) by the base (16).

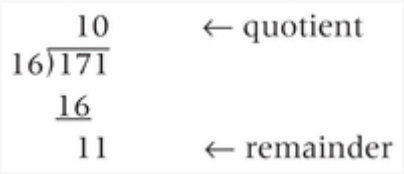


Fig 0.12b. (continued) Long division of 171 $(\text{quotient from previous}) \div 16$

The remainder (11) is the next digit to the left in the answer, which is represented by the digit B. Now the answer so far is BC. Since the quotient is not zero, we divide it (10) by the base (16).

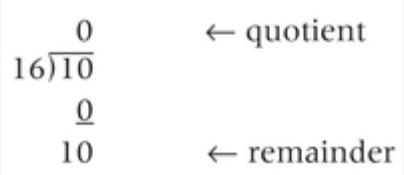


Fig 0.12c. (continued) Long division of 10 $(\text{quotient from previous}) \div 16$

The remainder (10) is the next digit to the left in the answer, which is represented by the digit A. Now the answer is ABC. Since the quotient is zero, we are finished, and the final answer is ABC.

0.7 Binary System and Memory

Modern computers are binary machines. Numbers within the computer are represented in binary form. In fact, all information is somehow represented using binary values. The reason is that each memory location within a computer contains either a low-voltage signal or a high-voltage signal. Because each location can have only one of two states, it is logical to equate those states to 0 and 1. A low-voltage signal is equated with a 0 (or *False*), and a high-voltage signal is equated with a 1 (or *True*). Note that a memory location cannot be empty: It must contain either a 0 or a 1.

Each storage unit is called a **binary digit** (0 or 1), or **bit** for short, and bits are grouped together into **bytes** (8 bits). Modern computers are often 64-bit machines.

> **Binary digit or bit** A digit in the binary number system; a 0 or a 1
>
> **Byte** Eight binary digits

Information such as numbers and letters must be represented in binary form to be stored in memory. The representation scheme used to store information is called a *code*. The next section examines a code for storing unsigned integers.

0.8 Unsigned Integer

The *unsigned* integer representation is for integers that are always non-negative. We have seen earlier that binary numbers are represented as 0s and 1s. Therefore, an *unsigned binary integer* is a positive integer represented by 0s and 1.

The **range** of an unsigned integer depends on how the computing system architecture or even the programming language. For the sake of explanation let's assume that we are dealing with a 7-bit system. The decimal number 22 can be represented as $001\ 0110$ on a 7-bit system. Note that the two leading 0s are still necessary. Therefore, a sequence of all 0s ($000\ 0000$) represents the smallest unsigned value (0 in decimal), a sequence of all 1s ($111\ 1111$) represents the largest ($2^7 - 1 = 127$ in decimal).

0.9 The Carry Bit

Earlier we have seen how arithmetic operations like addition and subtraction can be done on binary numbers. In the event that the value exceeds the base number, we just have to 'carry over' the to next column. Theoretically, this can be done indefinitely. However, computer memory is limited. As mentioned previously, a 7-bit system can have a maximum binary value of $111\ 1111$. So what happens when we add 1 to $111\ 1111$ in a 7-bit system? If we proceed with the addition, the expected value will be $1000\ 0000$ in binary. However, since we are dealing with a 7-bit system, it will only store the 7 rightmost bits and will incorrectly evaluate to $000\ 0000$ which is 0 in decimal.

To flag this condition, the CPU contains a special bit called the *carry bit*, denoted by the letter *C*. When two binary numbers are added, if the sum of the leftmost column (called the *most significant bit*) produces a carry, then *C* is set to 1. Otherwise *C* is cleared to 0. In other words, *C* always contains the carry from the leftmost column.

0.10 Two's Complement Binary Representation

The unsigned binary representation works for non negative integers only. If a computer is to process negative integers, it must use a different representation.

Suppose you have a 6-bit system and you want to store the decimal number -5, how would you represent the number with only 0s and 1s?

A simple solution to this problem is to reserve the first box in the system to indicate the sign. Thus, the six-bit system will have two parts—a one-bit sign and a five-bit magnitude, as shown in Fig 0.13.

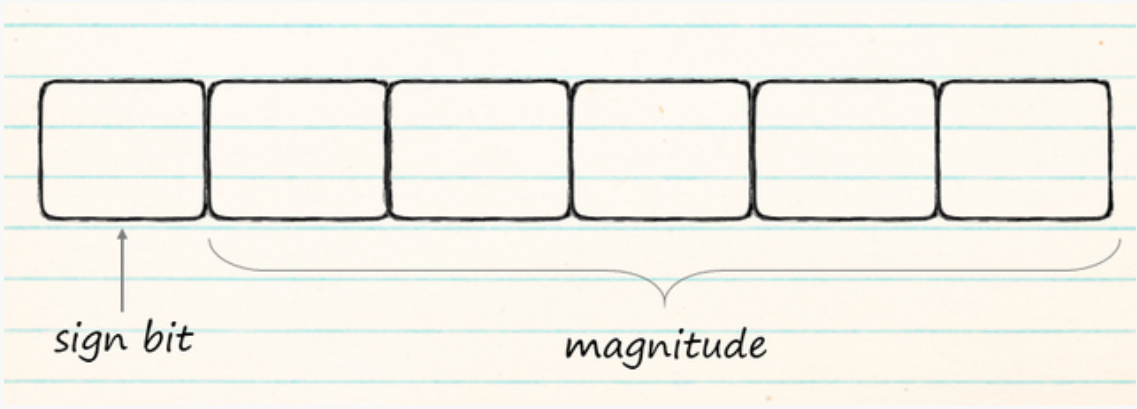


Fig 0.13. Structure of a 6-bit signed integer

Because the sign bit must be 0 or 1, one possibility is to let a 0 sign bit indicate a positive number and a 1 sign bit indicate a negative number. Then $+5$ could be represented as $00\ 0101$ and -5 could be represented as $10\ 0101$.

In this code, the magnitude portion for $+5$ and -5 would be identical i.e. $0\ 0101$. Only the sign bits would differ.

However, the problem is then when $+5$ is added to -5 in decimal, we should get 0 but adding $00\ 0101$ and $10\ 0101$ in binary will result to $10\ 1010$ which is definitely not zero.

$$\begin{array}{ll} & 00\ 0101 \\ \text{ADD} & 10\ 0101 \\ \hline \text{C} = 0 & 10\ 1010 \end{array}$$

Fig 0.14a. Sum of $00\ 0101$ ($+5$ in decimal) and $10\ 0101$ (-5 in decimal)

It would be much more convenient if the hardware of the CPU could add the numbers for $+5$ and -5, complete with sign bits using the ordinary rules for unsigned binary addition, and get zero in both decimal and binary representations.

The **two's complement** binary representation has that property. The positive numbers have a 0 sign bit and a magnitude as in the unsigned binary representation. For example, the decimal number $+5$ is still represented as $00\ 0101$ in binary, but the representation of -5 is **not** $10\ 0101$. Instead, it is $11\ 1011$ because adding $+5$ to -5 gives 0 in both decimal and the 6-bit system.

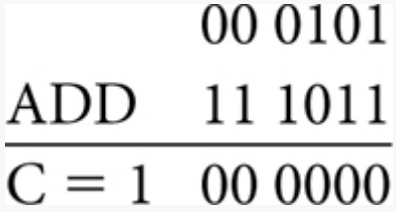


Fig 0.14b. Sum of $00\ 0101$ ($+5$ in decimal) and $11\ 1011$ (-5 in decimal)

Note that the 6-bit sum is all 0s.

Under the rules of binary addition for a six-bit system, the number $11\ 1011$ is called the *additive inverse* of 00 0101. The operation of finding the additive inverse is referred to as *negation*, abbreviated NEG. To negate a number is also called *taking its two's complement*.

All we need now is the rule for taking the two's complement of a number. A simple rule is based on the *ones' complement*, which is simply the binary sequence with all the 1's changed to 0's and all the 0's changed to 1's. The ones' complement is also called the NOT operation.

For example, the ones' complement of $00\ 0101$ in a 6-bit system is

$$\text{NOT}\ \ 00\ 0101 = 11\ 1010$$

A clue to finding the rule for two's complement is to note the effect of adding a number to its ones' complement. Because 1 plus 0 is 1, and 0 plus 1 is 1, any number, when added to its ones' complement, will produce a sequence of all 1's. But then, adding a single 1 to a number of all 1's produces a number of all 0's.

Example, adding $00\ 0101$ to its ones' complement 111010, produces all 1s.

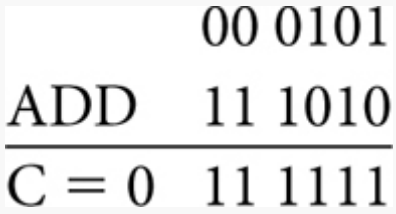


Fig 0.15. Adding $00\ 0101$ to its ones' complement

Then, adding 1 to this produces all 0s.

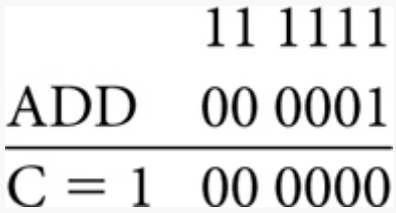


Fig 0.16. Adding 1 to $11\ 1111$ produces 0 in 6-bit system

In other words, adding a number to its ones' complement plus 1 gives all 0s. So, the two's complement of a binary number must be found by adding 1 to its ones' complement.

For example, to find the two's complement of $00\ 0101$, add 1 to its ones' complement.

$$\begin{array}{ll} \text{NOT} & 00\ 0101 = 11\ 1010 \\ & \\ & 11\ 1010 \\ \text{ADD} & \underline{00\ 0001} \\ & 11\ 1011 \end{array}$$

Fig 0.17. Finding two's complement of $00\ 0101$

The two's complement of 00 0101 is therefore 11 1011. That is,

$$\text{NEG}\ \ 00\ 0101 = 11\ 1011$$

Recall that $11\ 1011$ is indeed the negative of $00\ 0101$ because they add to 0 as shown.

The general rule for negating a number regardless of how many bits the number contains is also known as the two's complement rule.

> The two's complement of a number is 1 plus its ones' complement.

Or, in terms of the NEG and NOT operations,

$$\text{NEG}\ \ x = \text{NOT}\ \ x + 1$$

In our familiar decimal system, if you take the negative of a value that is already negative, you get a positive value. Algebraically, $-(-x) = x$, where x is some positive value. If the rule for taking the two's complement is to be useful, the two's complement of a negative value should be the corresponding positive value.

Therefore, if we take the two's complement of -5, we will get back $+5$ in binary.

$$\begin{array}{ll} \text{NOT} & 11\ 1011 = 00\ 0100 \\ & \\ & 00\ 0100 \\ \text{ADD} & \underline{00\ 0001} \\ & 00\ 0101 \end{array}$$

Fig 0.18. Finding two's complement of $11\ 1011$ (-5 in decimal)

0.11 Two's Complement Range

Suppose you have a 4-bit system to store integers in two's complement representation. What is the range of integers for this system? The positive integer with the greatest magnitude is 0111 ($+7$ in decimal). It cannot be 1111 as in unsigned binary because the first bit is reserved for the sign and must be 0. In unsigned binary, you can store numbers as high as $+15$ in decimal with only four bits. All four bits are used for the magnitude. In two's complement representation, you can store numbers only as high as $+7$ in decimal, because only three bits are reserved for the magnitude.

What is the negative number with the greatest magnitude? The answer to this question might not be obvious. The table below shows the result of taking the two's complement of each positive number up to +7. What pattern do you see in the table?

Decimal	Binary
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111

Notice that the two's complement operation automatically produces a 1 in the sign bit of the negative numbers, as it should. Even numbers still end in 0, and odd numbers end in 1.

Also, -5 is obtained from -6 by adding 1 to -6 in binary, as you would expect. Similarly, -6 is obtained from -7 by adding 1 to -7 in binary. We can squeeze one more negative integer out of our four bits by including -8. When you add 1 to -8 in binary, you get -7. The number -8 should therefore be represented as 1000. The table below shows the complete table for signed integers assuming a four-bit memory system.

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

The number -8 in decimal has a peculiar property not shared by any of the other negative integers. If you take the two's complement of -7, you get $+7$, as follows:

$$\begin{array}{ll} \text{NOT} & 1001 = 0110 \\ & \\ & 0110 \\ \text{ADD} & \underline{0001} \\ & 0111 \end{array}$$

Fig 0.19. Finding two's complement of 1001 (-7 in decimal)

But if you take the two's complement of -8, you get -8 back again:

$$\begin{array}{ll} \text{NOT} & 1000 = 0111 \\ & \\ & 0111 \\ \text{ADD} & \underline{0001} \\ & 1000 \end{array}$$

Fig 0.20. Finding two's complement of 1000 (-8 in decimal)

This property exists because there is no way to represent $+8$ with only four bits.

We have determined the range of numbers for a four-bit system with two's complement binary representation. It is 1000 to 0111 in binary (or -8 to $+7$ in decimal).

The same patterns hold regardless of how many bits that the system can contain. The largest positive integer is a single 0 followed by all 1s. The negative integer with the largest magnitude is a single 1 followed by all 0s. Its magnitude is 1 greater than the magnitude of the largest positive integer. The number -1 in decimal is represented as all 1s.

0.11 The Number Line

Another way of viewing binary representation is with the number line. Fig 0.21 shows the number line for a three-bit system with unsigned binary representation. Eight numbers are represented.

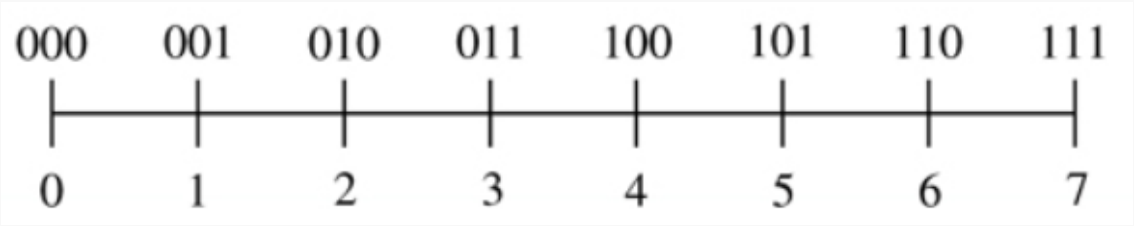


Fig 0.21. The number line for a 3-bit unsigned system

Addition is done by moving to the right on the number line. For example, to add 4 and 3, start with 4 and move three positions to the right to get 7. If you try to add 6 and 3 on the number line, you will fall off the right end. If you do it in binary, you will get an incorrect result because the answer is out of range:

$$\begin{array}{rr} & 110 \\ \text{ADD} & 011 \\ \hline \text{C} = 1 & 001 \end{array}$$

Fig 0.22. Adding 110 to 011 on the number line in a 3-bit unsigned system

The two's complement number line comes from the unsigned number line by breaking it between 3 and 4 and shifting the right part to the left side.

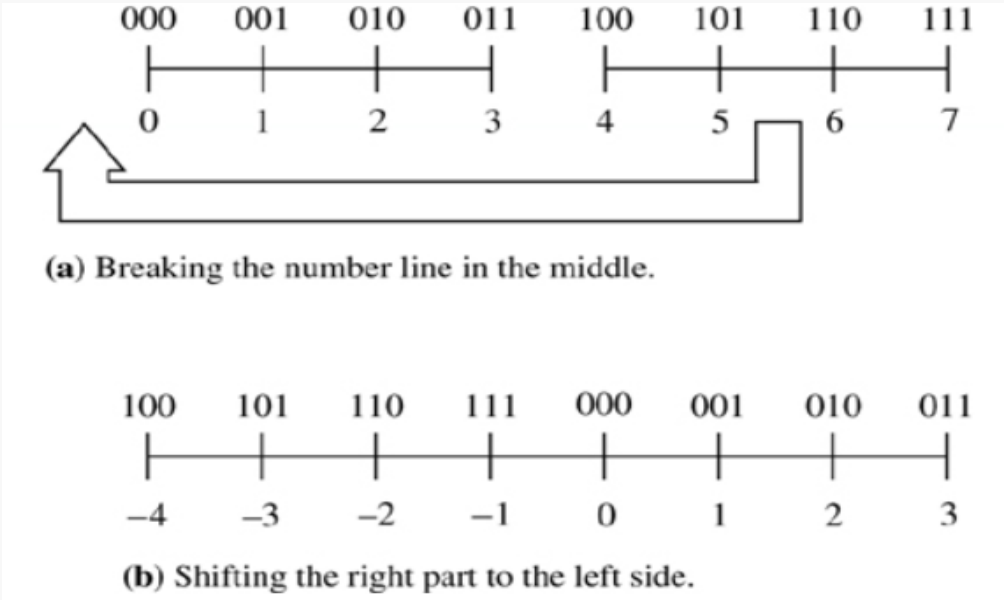


Fig 0.23. The number line for a 3-bit two's complement system

Fig 0.23 shows that the binary number 111 is now adjacent to 000, and what used to be $+7$ in decimal is now -1 in decimal.

Addition is still performed by moving to the right on the number line, even if you pass through 0. To add –2 and 3, start with –2 and move three positions to the right to get 1. If you do it in binary, the answer is in range and correct:

$$\begin{array}{rr} & 110 \\ \text{ADD} & 011 \\ \hline \text{C} = 1 & 001 \end{array}$$

Fig 0.24. Adding 110 to 011 on the number line in a 3-bit two's complement system

These bits are identical to those for $6 + 3$ in unsigned binary. Notice that the carry bit is 1, even though the answer is in range. With two's complement representation, the carry bit no longer indicates whether the result of the addition is in range.

Sometimes you can avoid the binary representation altogether by considering the shifted number line entirely in decimal. Fig 3.25 shows the two's complement number line with the binary number replaced by its unsigned decimal equivalent. In this example, there are three bits in each memory location. Thus, there are 2^3, or 8, possible numbers.

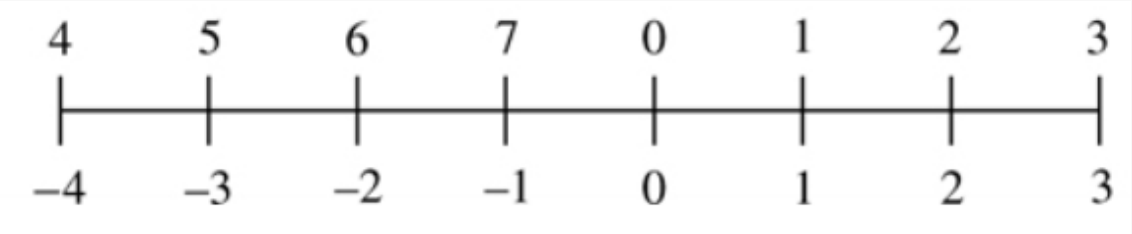


Fig 0.25. The two's complement number line with unsigned decimals

Now the unsigned and signed numbers are the same from 0 up to 3. Furthermore, you can get the signed negative numbers from the unsigned numbers by subtracting 8:

$$\begin{aligned} 7 - 8 &= -1 \\ 6 - 8 &= -2 \\ 5 - 8 &= -3 \\ 4 - 8 &= -4 \end{aligned}$$

Suppose you have an eight-bit system. There are 2^8, or 256, possible integer values. The non-negative numbers go from 0 to 127. Assuming two's complement binary representation, what do you get if you add 97 and 45? In unsigned binary, the sum is $97 + 45 = 142$ (decimal, unsigned). But in two's complement binary, the sum is $142 - 256 = -114$ (decimal, signed).

Notice that we get this result by avoiding the binary representation altogether. To verify the result, first convert 97 and 45 to binary and add:

$$97_{10} = 0110\ 0001_2$$
$$45_{10} = 0010\ 1101_2$$

$$\begin{array}{ll} & 0110\ 0001 \\ \text{ADD} & 0010\ 1101 \\ \hline \text{C} = 0 & 1000\ 1110 \end{array}$$

Fig 0.26. Adding 97 to 45 in binary.

This is a negative number because of the 1 in the sign bit. And now, to determine its magnitude:

$$\text{NEG}\ 1000\ 1110 = 0111\ 0010 = 114\ (\text{in decimal})$$

This produces the expected result.

0.12 The Overflow Bit

An important property of binary data representation is the absence of a type associated with a value. In the previous example, the binary $1000\ 1110$, when interpreted as an unsigned number, is 142 in binary, but when interpreted in two's complement representation is -114 in decimal. Although the value of the bit pattern depends on its type, whether unsigned or two's complement, the hardware makes no distinction between the two types. It stores only the bit pattern.

When the CPU adds the contents of two memory cells, it uses the rules for binary addition on the bit sequences, regardless of their types. In unsigned binary, if the sum is out of range, the hardware simply stores the (incorrect) result, sets the carry bit *C* accordingly, and goes on. It is up to the software to examine the *C* bit after the addition to see if a carry out occurred from the most significant column and to take appropriate action if necessary.

> *The C bit detects overflow for unsigned integers.*

We noted above that in two's complement binary representation, the carry bit no longer indicates whether a sum is in range or out of range. An *overflow condition* occurs when the result of an operation is out of range. To flag this condition for signed numbers, the CPU contains another special bit called the *overflow bit*, denoted by the letter *V*. When the CPU adds two binary integers, if their sum is out of range when interpreted in the two's complement representation, then *V* is set to 1. Otherwise *V* is cleared to 0.

> *The V bit detects overflow for signed integers.*

The CPU performs the same addition operation regardless of the interpretation of the bit pattern. As with the *C* bit, the CPU does not stop if a two's complement overflow occurs. It sets the *V* bit and continues with its next task. It is up to the software to examine the *V* bit after the addition.

Fig 0.27 shows some examples in six-bit systems, demonstrating the effects of the the carry bit *C* and the overflow bit *V*.

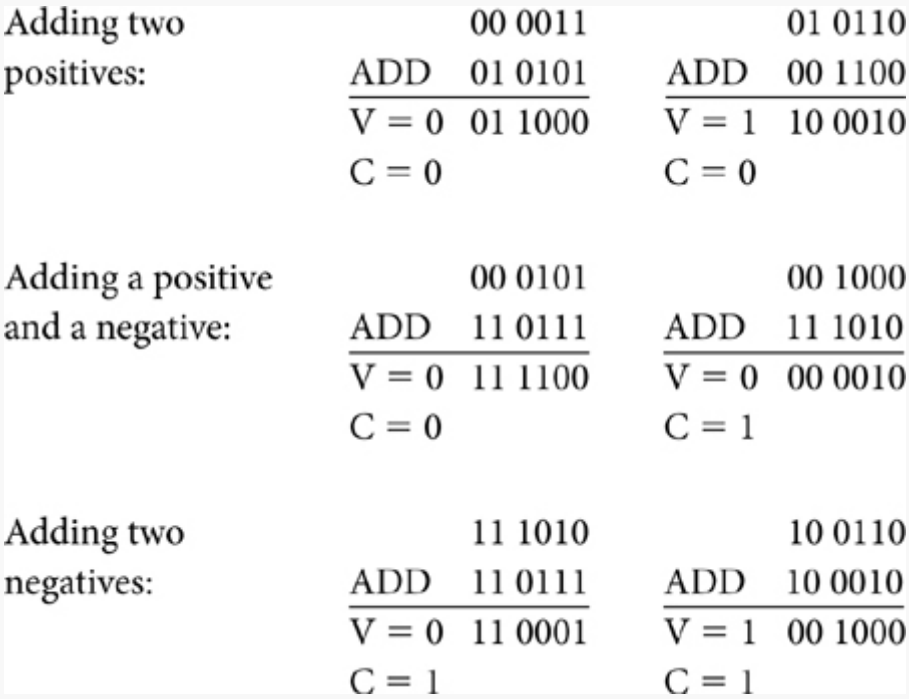


Fig 0.27. All combinations of values possible for *V* and *C*

How can you tell if an overflow condition will occur? One way would be to convert the two numbers to decimal, add them, and see if their sum is outside the range as written in decimal. If so, an overflow has occurred.

The hardware detects an overflow by comparing the carry-in to the sign bit with the *C* bit. If they are different, an overflow has occurred, and *V* gets 1. If they are the same, *V* gets 0.

Instead of comparing the carry-in to the sign bit with *C*, you can tell directly by inspecting the signs of the numbers and the sum. If you add two positive numbers and get a negative sum, or if you add two negative numbers and get a positive sum, then an overflow occurred. It is not possible to get an overflow by adding a positive number and a negative number.

0.13 The Negative and Zero Bits

In addition to the *C* bit, which detects an overflow condition for unsigned integers, and the *V* bit, which detects an overflow condition for signed integers, the CPU maintains two other bits that the software can test after it performs an operation. They are the *N* bit, for detecting a negative result, and the *Z* bit, for detecting a zero result. In summary, the functions of these four status bits are:

- $N = 1$ if the result is negative. Otherwise, $N = 0$.
- $Z = 1$ if the result is all zeros. Otherwise, $Z = 0$
- $V = 1$ if a signed integer overflow occurred. Otherwise, $V = 0$
- $C = 1$ if an unsigned integer overflow occurred. Otherwise, $C = 0$

The *N* bit is easy for the hardware to determine, as it is simply a copy of the sign bit. It takes a little more work for the hardware to determine the *Z* bit, because it must determine if every bit of the result is zero.

Fig 0.28 shows some examples of addition that show the effect of all four status bits on the result.

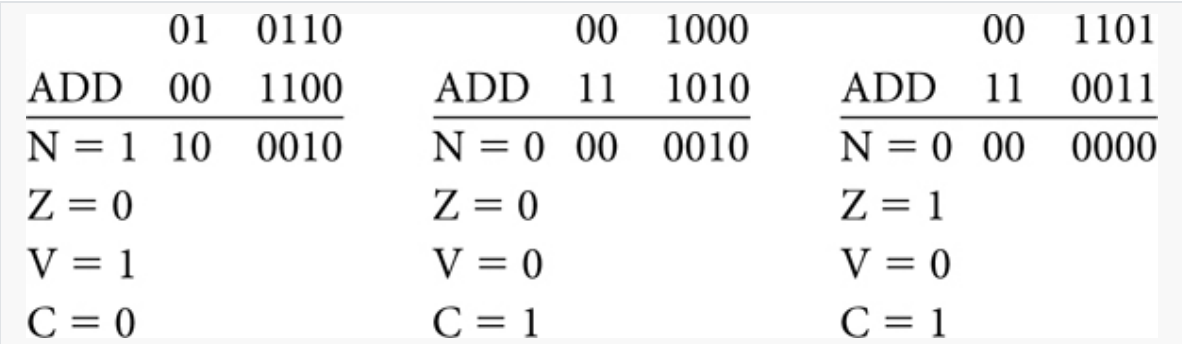


Fig 0.28. Examples of addition showing effects of all four status bits

0.14 Binary Arithmetic

Because computers use binary representation, programmers who write great code often have to work with binary (and hexadecimal) values. Often, when writing code, you may need to manually operate on two binary values in order to use the result in your source code. Although calculators are available to compute such results, you should be able to perform simple arithmetic operations on binary operands by hand.

Arithmetic operations on binary values, are actually easier than decimal arithmetic. Knowing how to manually compute binary arithmetic results is essential because several important algorithms use these operations (or variants of them). Therefore, the next several subsections describe how to manually add, subtract, multiply, and divide binary values, and how to perform various logical operations on them.

0.14.1 Adding Binary Values

Earlier we have seen how we can add 2 binary values together. The steps to adding binary values is the same as adding decimal numbers. To summarize, there are 8 rules to note when adding two binary values:

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 0 + C$
- $C + 0 + 0 = 1$
- $C + 0 + 1 = 0 + C$
- $C + 1 + 0 = 0 + C$
- $C + 1 + 1 = 1 + C$

  Note: C is the carry bit.

As concrete example,

```
          0101
        + 0011
        ------

Step 1: Add the rightmost bits (1 + 1 = 0 + C).

            c
          0101
        + 0011
        ------
             0

Step 2: Add the carry plus the bits in bit position one (carry + 0 + 1 = 0 + C).

           c
          0101
        + 0011
        -------
            00
Step 3: Add the carry plus the bits in bit position two (carry + 1 + 0 = 0 + C).

          c
          0101
        + 0011
        ------
           000

Step 4: Add the carry plus the bits in bit position three (C + 0 + 0 = 1).

          0101
        + 0011
        ------
          1000
```

Here are more examples:

```
  1100 1101       1001 1111       0111 0111
+ 0011 1011     + 0001 0001     + 0000 1001
-----------     -----------     -----------
1 0000 1000       1011 0000       1000 0000
```

0.14.2 Subtracting Binary Values

Subtracting binary values is also similar to adding. Notice that subtracting is actually adding of the first binary value to the NEG or two's complement of the second value. Thus, the same rules apply to subtracting and adding.

Consider the following example:

```
          0101
        - 0011
        ------

Step 1: Determine the NEG or two's complement of the second value.

      NOT 0101
        ------
          1010

          1010
        + 0001
        ------
          1011

Step 2: Add first value to the two's complement of the second value.


          0101
        + 1011
        -------

Step 3: Add the rightmost bit i.e. bit position 0 (1 + 1 = 0 + C)

            C
          0101
        + 1011
        ------
             0

```

```
Step 4: Add the carry plus the bits in bit position one (C + 0 + 1 = 0 + C).

           C
          0101
        + 1011
        ------
            00

Step 5: Add the carry plus the bits in bit position two (C + 1 + 0 = 0 + C).

          C
          0101
        + 1011
        ------
           000

Step 6: Add the carry plus the bits in bit position three (C + 0 + 1 = 0 + C).

          C
          0101
        + 1011
        ------
        1 0000

```

Here are more examples:

```
  1100 1101       1001 1111       0111 0111
- 0011 1011     - 0001 0001     - 0000 1001
-----------     -----------     -----------
  1001 0010       1000 1110       0110 1110
```

0.14.3 Multiplying Binary Values

Multiplication of binary numbers works the same way as decimal multiplication. However, it involves only ones and zeros:

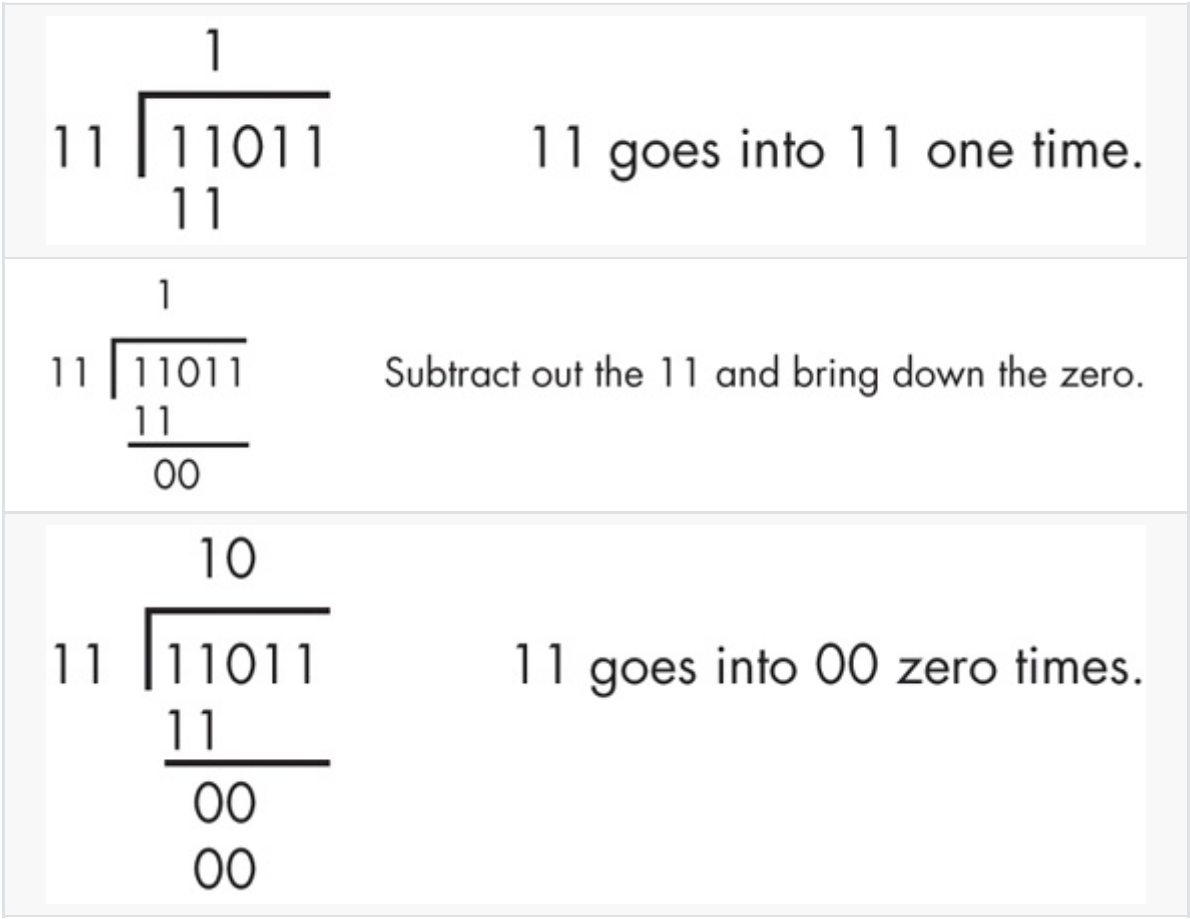
- $0 \times 0 = 0$
- $0 \times 1 = 0$
- $1 \times 0 = 0$
- $1 \times 1 = 1$

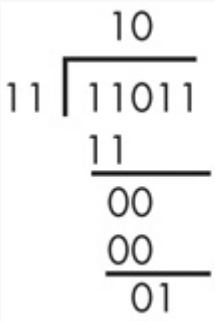
```
      1010 (multiplicand)
   ×  0101 (multiplier)
   -------

Step 1: Multiply the rightmost bit of the multiplier times the multiplicand.

      1010
   ×  0101
   -------
      1010    (1 × 1010)

Step 2: Multiply bit one of the multiplier times the multiplicand.

      1010
   ×  0101
   -------
      1010    (1 × 1010)
     0000     (0 × 1010)
   -------
     01010    (partial sum)

Step 3: Multiply bit two of the multiplier times the multiplicand.

      1010
   ×  0101
   -------
    001010    (previous partial sum)
    1010      (1 × 1010)
   -------
    110010    (partial sum)

Step 4: Multiply bit three of the multiplier times the multiplicand.

      1010
   ×  0101
   -------
    110010    (previous partial sum)
   0000       (0 × 1010)
   -------
   0110010    (product)
```

0.14.3 Dividing Binary Values

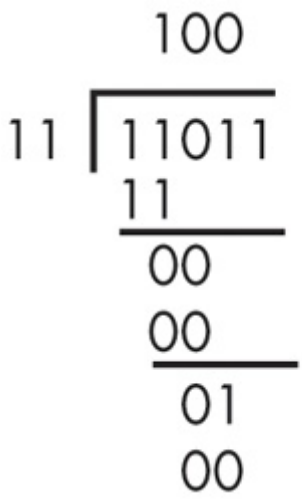
In chapter 0.6, we have seen the long division algorithm when converting base. Long division in binary also works the same way. In fact, it is easier as the divisor goes into remainder exactly one or zero times.

Consider the division of $1\ 1011$ (27 in decimal) by 11 (3 in decimal):

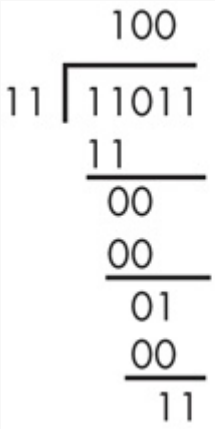




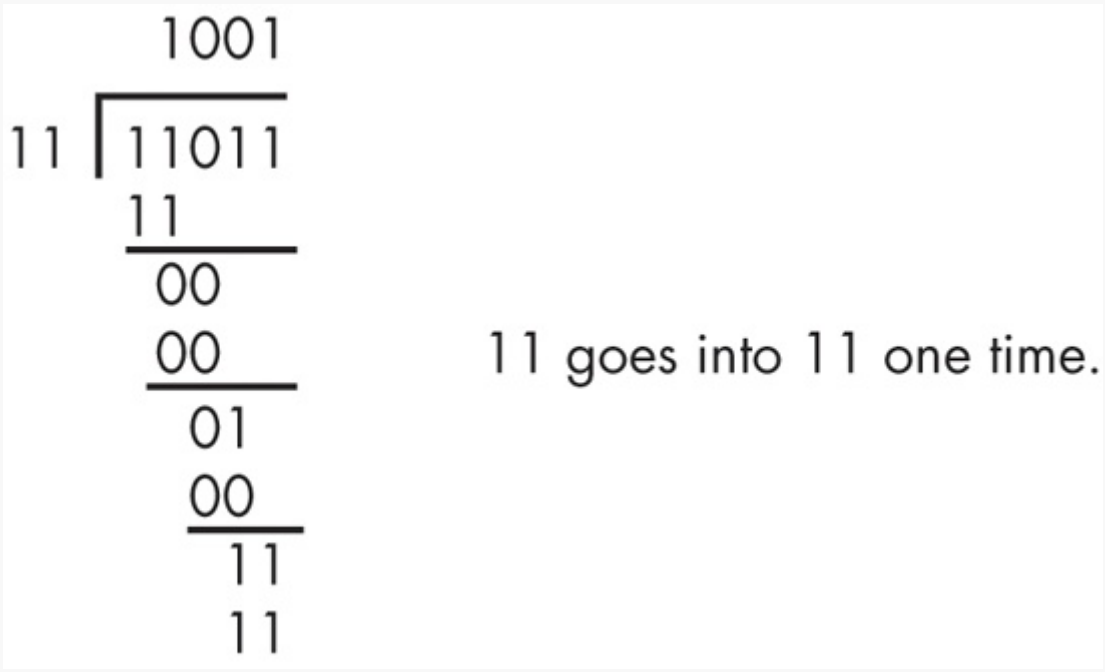
Subtract out the zero and bring down the one.



11 goes into 01 zero times.



Subtract out the zero and bring down the one.



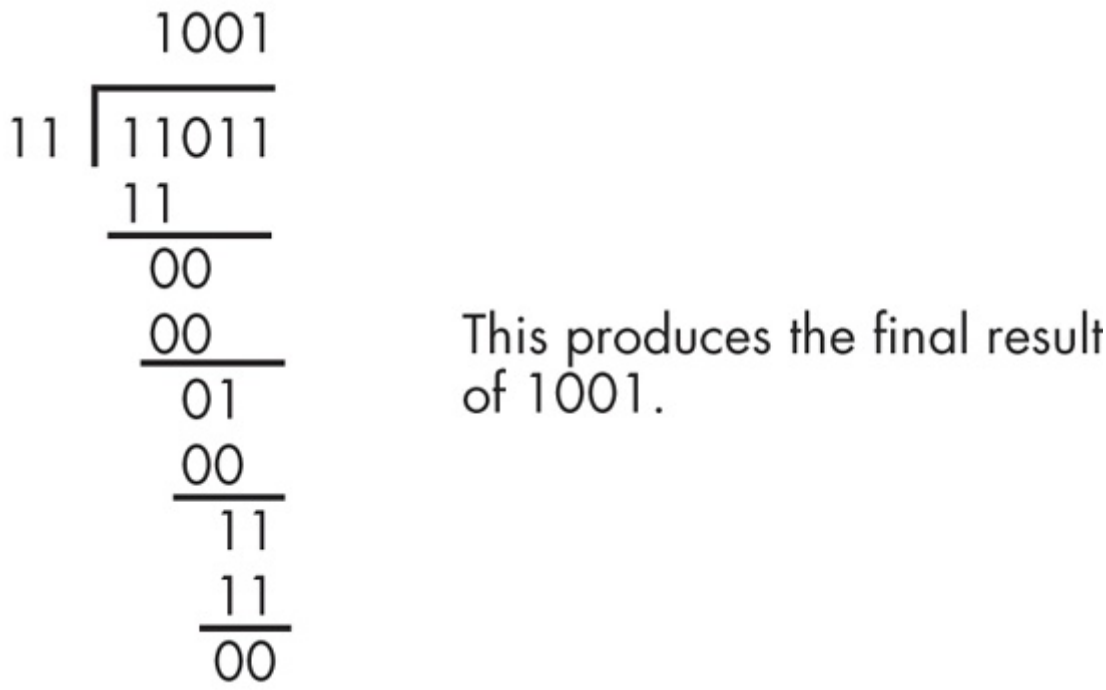


Fig 0.28. Long Division in binary

0.15 Logical Operations on Bits

There are four main logical operations we'll need to perform binary numbers: AND, OR, XOR (exclusive-or), and NOT.

The logical AND, OR, and XOR operations accept two single-bit operands. The following tables show the *truth tables* for the AND, OR, and XOR operations.

A truth table is just like the multiplication tables you encountered in elementary school. The values in the left column correspond to the left operand of the operation. The values in the top row correspond to the right operand of the operation. The value located at the intersection of the row and column (for a particular pair of input values) is the result.

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

The truth table is named as such because a combination of conditions gives a *True* (represented by 1) or *False* (represented by 0) depending on the logical operation.

For example, the statement "the number 8 larger than 7 AND smaller than 9?" considered a True statement as the first condition that 8 is larger than 7 is true, and the second condition that 8 is smaller than 9 is also true.

On the other hand, the statement "the number 8 is even AND larger than 9" is considered a False statement even if the first statement that 8 is even is true. This is because the second statement that 8 is larger than 9 is false. For the entire statement to be True, both conditions need to be True.

In plain English, the logical AND operation translates as, "If the first operand is one and the second operand is one, the result is one; otherwise the result is zero." We could also state this as "If either or both operands are zero, the result is zero." The logical AND operation is useful for forcing a zero result. If one of the operands is zero, the result is always zero regardless of the value of the other operand. If one of the operands contains one, then the result is the value of the other operand.

Colloquially, the logical OR operation is, "If the first operand or the second operand (or both) is one, the result is one; otherwise the result is zero." This is also known as the *inclusive-OR* operation. If one of the operands to the logical-OR operation is one, the result is always one. If an operand is zero, the result is always the value of the other operand.

In English, the logical XOR operation is, "If the first or second operand, but not both, is one, the result is one; otherwise the result is zero." If one of the operands is a one, the result is always the *inverse* of the other operand.

We have also seen the logical NOT operation. The logical NOT operation is **unary** (meaning it accepts only one operand). The truth table for the NOT operation appears in the table below. This operator simply inverts (reverses) the value of its operand from 0 to 1 and vice versa.

NOT	0	1
	1	0

0.16 Logical Operations on Binary Numbers

We need to extend the definition of logical operations beyond single-bit operands. We can easily extend logical operations to operate on a *bit-by-bit* (or *bitwise*) basis. Given two binary values, a bitwise logical function operates on bit position zero of both operands producing bit position zero of the result; it operates on bit position one of both operands producing bit position one of the result, and so on. For example, if you want to compute the bitwise logical AND of two 8-bit numbers, you would logically AND each pair of bits in the two numbers:

```
    1011 0101
AND 1110 1110
   ----------
    1010 0100
```

This bit-by-bit execution also applies to the other logical operations, as well. The ability to force bits to zero or one using the logical AND and OR operations, and the ability to invert bits using the logical XOR operation, is very important when working with strings of bits (such as binary numbers). These operations let you selectively manipulate certain bits within a value while leaving other bits unaffected.

For example, if you have an 8-bit binary value X and you want to guarantee that bit positions four through seven contain zeros, you could logically AND the value X with the binary value $0000\ 1111$. This bitwise logical AND operation would force the four leftmost bits of X to zero and leave the four rightmost bits of X unchanged.

Likewise, you could force the rightmost bit of *X* to one and invert bit position two of X by logically OR the value X with $0000\ 0001$ and then logically XOR the value X with $0000\ 0100$. Using the logical AND, OR, and XOR operations to manipulate bit strings in this fashion is known as **masking** bits. We use the term *masking* because we can use certain values (one for AND, zero for OR and XOR) to "mask out" or "mask in" certain bits in an operand while forcing other bits to zero, one, or their inverse. This has plenty of application in computer science especially in cryptography, computer vision, data processing, etc.

Several languages provide operators that let you compute the bitwise AND, OR, XOR, and NOT of their operands. The Python language uses the ampersand (`&`) operator for bitwise AND, the pipe (`|`) operator for bitwise OR, the caret (`^`) operator for bitwise XOR, and the tilde (`~`) operator for bitwise NOT.

```
'''Bitwise operations in Python'''
i = j & k    # Bitwise AND
i = j | k    # Bitwise OR
i = j ^ k    # Bitwise XOR
i = ~j       # Bitwise NOT
```

0.17 Floating-Point Representation

The numeric representations described in the previous sections of this chapter are for integer values. How then do we represent the rest of the real numbers like -0.5, $0.33333\ldots$, 1.2345×10^{6789}, π, etc.?

There are several mechanisms by which strings of digits can represent numbers.

In common mathematical notation, the digit string can be of any length, and location of the radix point, also know as decimal point or binary point, is indicated by placing an explicit dot.

In scientific notation, the given number is scaled by a power of 10, so that it lies within a certain range - typically between 1 and 10, with the radix point appearing immediately after the first digit e.g. 6.02×10^{23}.

Floating-point representation is similar in concept to scientific notation but in binary.

0.17.1 Binary Fractions

Firstly, let's discuss about fractions in binary. Binary fractions have a binary point, which is the base-2 version of the base-10 decimal point. Fig 0.28(a) shows the place values for 101.011 in binary. The bits to the left of the binary point have the same place values as the corresponding bits in unsigned binary representation. Starting with the $\frac{1}{2}$'s place to the right of the binary point, each place has a value one-half as great as the previous place value. Figure 0.28(b) shows the addition that produces the 5.375 value in decimal.

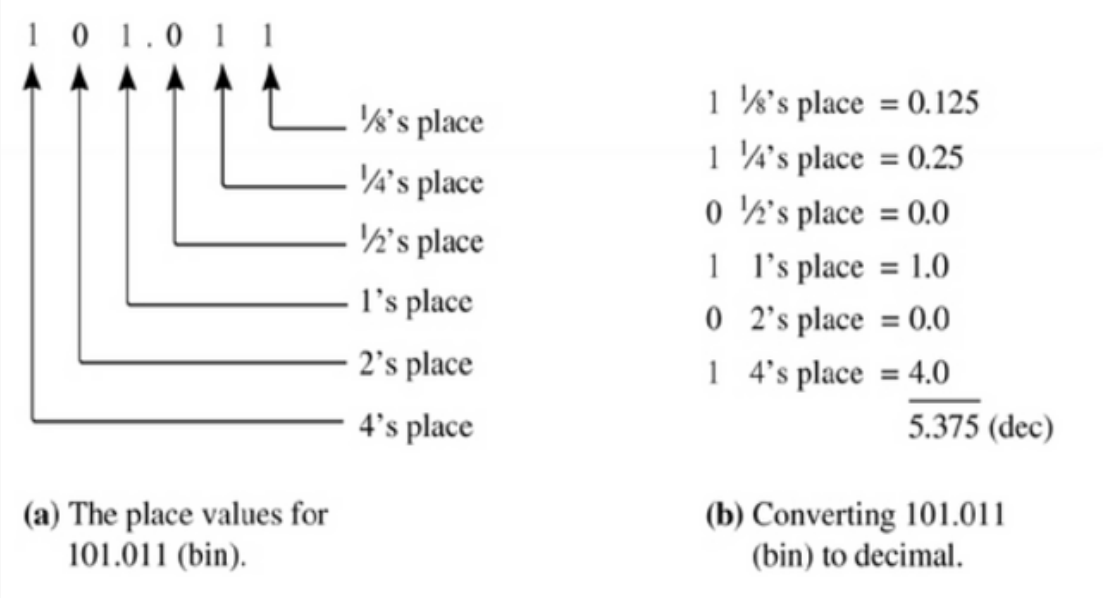


Fig 0.28. Converting from binary to decimal

Fig 0.29 shows the polynomial representation of numbers with fractional parts. The value of the bit to the left of the radix point is always the base to the zeroth power, which is always 1. The next significant place to the left is the base to the first power, which is the value of the base itself. The value of the bit to the right of the radix point is the base to the power -1. The next significant place to the right is the base to the power -2. The value of each place to the right is $\frac{1}{base}$ times the value of the place on its left.

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

(a) The binary number 101.011.

$$5 \times 10^2 + 0 \times 10^1 + 6 \times 10^0 + 7 \times 10^{-1} + 2 \times 10^{-2} + 1 \times 10^{-3}$$

(b) The decimal number 506.721.

Fig 0.29. The polynomial representation of floating-point numbers

Determining the decimal value of a binary fraction requires two steps. First, convert the bits to the left of the binary point using the technique seen earlier for converting unsigned binary values (refer to Chapter 0.5). Then, use the algorithm of successive doubling to convert the bits to the right of the binary point.

Fig 0.30 shows the conversion of 6.5859375 decimal to binary. The conversion of the whole part gives 110 in binary to the left of the binary point. To convert the fractional part, write the digits to the right of the decimal point in the heading of the right column of the table. Double the fractional part, writing the digit to the left of the decimal point in the column on the left and the fractional part in the column on the right. Also note that the next time you double, do not include the whole number part. For example, the value 0.34375 comes from doubling 0.171875, not from doubling 1.171875. The digits on the left from top to bottom are the bits of the binary fractional part from left to right. So, $6.5859375_{10} = 110.1001011_2$.

6.5859375

↑

6 (dec) = 110 (bin)

(a) Convert the whole part.

	.5859375
1	.171875
0	.34375
0	.6875
1	.375
0	.75
1	.5
1	.0

(b) Convert the fractional part.

Fig 0.30. Converting from decimal to binary

The algorithm for converting the fractional part from decimal to binary is the mirror image of the algorithm for converting the whole part, from decimal to binary. In Chapter 0.6, we have seen how to convert the whole part. To convert the whole part to binary, we can use the algorithm of successive division by two. The bits you generate are the remainders of the division, and you generate them from right to left starting at the binary point. To convert the fractional part, you use the algorithm of successive multiplication by two. The bits you generate are the whole part of the multiplication, and you generate them from left to right starting at the binary point.

A number that can be represented with a finite number of digits in decimal may require an endless representation in binary.

The table below shows the conversion of decimal 0.2 to binary. The first doubling produces 0.4. A few more doublings produce 0.4 again. It is clear that the process will never terminate and that $0.2_{10} = 0.001100110011_2$ with the bit pattern 0011 endlessly repeating.

	.2
0	.4
0	.8
1	.6
1	.2
0	.4
0	.8
1	.6
⋮	⋮

Because all computer memory can store only a finite number of bits, the value 0.2 cannot be stored exactly and must be approximated. You should realize that if you add $0.2 + 0.2$ in a some language like C, you will probably not get 0.4 exactly because of the roundoff error inherent in the binary representation of the values. For that reason, good numeric software rarely tests two floating point numbers for strict equality. Instead, the software maintains a small but nonzero tolerance that represents how close two floating point values must be to be considered equal. If the tolerance is, say, 0.0001, then the numbers 1.38264 and 1.38267 would be considered equal because their difference, which is 0.00003, is less than the tolerance.

0.17.2 Excess Representations

Floating-point numbers are represented with a binary version of the scientific notation common with decimal numbers. A nonzero number is *normalized* if it is written in scientific notation with the first nonzero digit immediately to the left of the radix point. The number zero cannot be normalized because it does not have a first nonzero digit.

To illustrate this, the decimal number -328.4 is written in normalized form in scientific notation as -3.284×10^2. The effect of the exponent 2 as the power of 10 is to shift the decimal point two places to the right. Similarly, the binary number -10101.101 is written in normalized form in scientific notation as -1.0101101×2^4. The effect of the exponent 4 as the power of 2 is to shift the binary point four places to the right. On the hand, the binary number 0.00101101 is written in normalized form in scientific notation as 1.01101×2^{-3}. The effect of the exponent -3 as the power of 2 is to shift the binary point three places to the left.

In general, a floating point number can be positive or negative, and its exponent can be a positive or negative integer. Fig 0.31 shows a *cell* in memory that stores a floating point value. The cell is divided into three fields. The first field stores one bit for the sign of the number. The second field stores the bits representing the exponent of the normalized binary number. The third field, called the *significand* (also known as *mantissa* or *coefficient*), stores bits that represent the magnitude of the value.

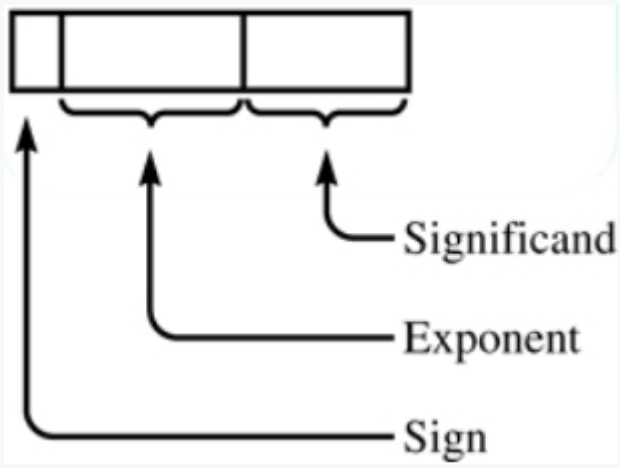


Fig 0.31. Storage for a floating-point value in a memory cell

The more bits stored in the exponent, the wider the range of floating point values. The more bits stored in the significand, the higher the precision of the representation. A common representation is an 8-bit cell for the exponent and a 23-bit cell for the significand. To present the concepts of the floating point format, the examples in this section use a 3-bit cell for the exponent and a 4-bit cell for the significand. These are unrealistically tiny cell sizes, but they help to illustrate the format without an unwieldy number of bits.

Any signed representation for integers could be used to store the exponent. You might think that two's complement binary representation would be used, because that is the representation that most computers use to store signed integers. However, two's complement is not used. Instead, a biased representation is used for a reason that will be explained later.

An example of a biased representation for a five-bit cell is excess-15. The range of numbers for the cell is -15 to 16 as written in decimal and 00000 to 11111 as written in binary. To convert from decimal to excess-15, you add 15 to the decimal value and then convert to binary as you would an unsigned number. To convert from excess-15 to decimal, you write the decimal value as if it were an unsigned number and subtract 15 from it. In excess-15, the first bit denotes whether a value is positive or negative. But unlike two's complement representation, 1 signifies a positive value, and 0 signifies a negative value.

For a five-bit system, to convert 5 from decimal to excess-15, add $5 + 15 = 20$. Then convert 20 to binary as if it were unsigned, 20 (dec) = 10100 (bin). Therefore, 5 (dec) = 10100 (excess-15). The first bit is 1, indicating a positive value.

To convert from excess-n to decimal, just mirror the steps. To convert 00011 from excess-15 to decimal, convert 00011 as an unsigned value, 00011 (bin) = 3 (dec). Then subtract decimal values $3 - 15 = -12$. So, 00011 (excess-15) = -12 (dec).

The table below shows the bit patterns for a three-bit cell that stores integers with excess-3 representation compared to two's complement representation. Each representation stores eight values. The excess-3 representation has a range of -3 to 4 (dec), while the two's complement representation has a range of -4 to 3 (dec).

Decimal	Two's Complement	Excess 3
-4	100	
-3	101	000
-2	110	001
-1	111	010
0	000	011
1	001	100
2	010	101
3	011	110
4		111

0.17.3 The Hidden Bit

Fig 0.31 shows one bit reserved for the sign of the number but no bit reserved for the binary point. A bit for the binary point is unnecessary because numbers are stored normalized, so the system can assume that the first 1 is to the left of the binary point. Furthermore, because there will always be a 1 to the left of the binary point, there is no need to store the leading 1 at all. To store a decimal value, first convert it to binary, write it in normalized scientific notation, store the exponent in excess representation, drop the leading 1, and store the remaining bits of the magnitude in the significand. The bit that is assumed to be to the left of the binary point but that is not stored explicitly is called the *hidden bit.*

Assuming a three-bit exponent using excess-3 and a four-bit significand, how is the number 3.375 stored? Converting the whole number part gives 3 (dec) = 11 (bin). Converting the fractional part gives 0.375 (dec) = 0.011 (bin). The complete binary number is 3.375 (dec) = 11.011 (bin), which is 1.1011×2^1 in normalized binary scientific notation. The number is positive, so the sign bit is 0. The exponent is 1 (dec) = 100 (excess-3) (referring to the previous table). Dropping the leading 1, the four bits to the right of the binary point are $.1011$. So, 3.375 is stored as $0100\ 1011$.

Of course, the hidden bit is assumed, not ignored. When you read a decimal floating point value from memory, the compiler assumes that the hidden bit is not stored. It generates code to insert the hidden bit before it performs any computation with the full number of bits. The floating point hardware even adds a few extra bits of precision called *guard digits* that it carries throughout the computation. After the computation, the system discards the guard digits and the assumed hidden bit and stores as many bits to the right of the binary point as the significand will hold.

Not storing the leading 1 allows for greater precision. In the previous example, the bits for the magnitude are 1.1011. Using a hidden bit, you drop the leading 1 and store $.1011$ in the four-bit significand. In a representation without a hidden bit, you would store the most significant bits, 1.011, in the four-bit significand and be forced to discard the least significant 0.0001 value. The result would be a value that only approximates the decimal value 3.375.

Because every memory cell has a finite number of bits, approximations are unavoidable even with a hidden bit. The system approximates by rounding off the least significant bits it must discard using a rule called “round to nearest, ties to even.” The table below shows how the rule works for decimal and binary numbers. You round off 23.499 to 23 because 23.499 is closer to 23 than it is to 24. Similarly, 23.501 is closer to 24 than it is to 23. However, 23.5 is just as close to 23 as it is to 24, which is a tie. It rounds to 24 because 24 is even. Similarly, the binary number 10111.1 is just as close to 10111 as it is to 11000, which is a tie. It rounds to 11000 because 11000 is even.

Decimal Rounding

Decimal	Decimal Rounded
23.499	23
23.5	24
23.501	24
24.499	24
24.5	24
24.501	25

Binary Rounding

Binary	Binary Rounded
10111.011	10111
10111.1	11000
10111.100	11000
11000.011	11000
11000.1	11000
11000.100	11000

Assuming a three-bit exponent using excess-3 and a four-bit significand, how is the number -13.75 stored? Converting the whole number part gives 13 (dec) = 1101 (bin). Converting the fractional part gives 0.75 (dec) = 0.11. The complete binary number is 13.75 (dec) = 1101.11 (bin), which is 1.10111×2^3 in normalized binary scientific notation. The number is negative, so the sign bit is 1. The exponent is 3 (dec) = 110 (excess-3). Dropping the leading 1, the five bits to the right of the binary point are $.10111$. However, only four bits can be stored in the significand. Furthermore, $.10111$ is just as close to $.1011$ as it is to $.1100$, and the tie rule is in effect. Because 1011 is odd and 1100 is even, round to $.1100$. So, -13.75 is stored as $1110\ 1100$.

0.17.4 Special Values

Some real values require special treatment. The most obvious is zero, which cannot be normalized because there is no 1 bit in its binary representation. You must set aside a special bit pattern for zero. Standard practice is to put all 0s in the exponent field and all 0s in the significand as well. What do you put for the sign? Most common is to have two representations for zero, one positive and one negative. For a three-bit exponent and four-bit significand, the bit patterns are $1000\ 0000$ (bin) = -0.0 (dec) and $0\ 000\ 0000$ (bin) = $+0.0$ (dec).

This solution for storing zero has ramifications for some other bit patterns. If the bit pattern for $+0.0$ were not special, then $0\ 000\ 0000$ would be interpreted with the hidden bit as 1.0000×2^{-3} (bin) = 0.125, the smallest positive value that could be stored had the value not been reserved for zero. If this pattern is reserved for zero, then the smallest positive value that can be stored is $0\ 000\ 0001 = 1.0001 \times 2^{-3}$ (bin) = 0.1328125 which is slightly larger. The negative number with the smallest possible magnitude is identical but with a 1 in the sign bit. The largest positive number that can be stored is the bit pattern with the largest exponent and the largest significand. The bit pattern for the largest value is $0\ 111\ 1111$ (bin) = $+31.0$ (dec).

Fig 0.32 shows the number line for the representation where zero is the only special value. As with integer representations, there is a limit to how large a value you can store. If you try to multiply 9.5 times 12.0, both of which are in range, the true value is 114.0, which is in the positive overflow region.

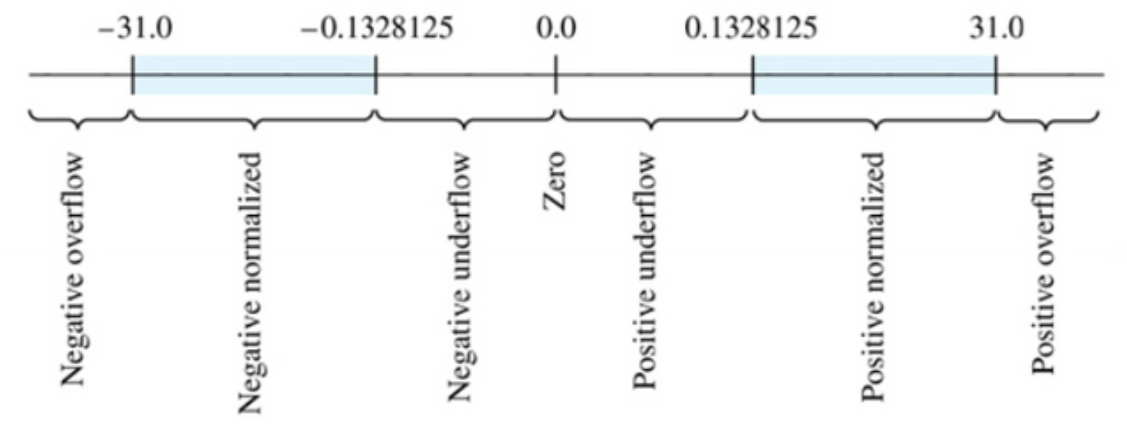


Fig 0.32. The real number line with zero as the only special value

Unlike integer values, however, the real number line has an underflow region. If you try to multiply 0.145 times 0.145, which are both in range, the true value is 0.021025, which is in the positive underflow region. The smallest positive value that can be stored is 0.132815.

Numeric calculations with approximate floating point values need to have results that are consistent with what would be expected when calculations are done with exact precision. For example, suppose you multiply 9.5 and 12.0. What should be stored for the result? Suppose you store the largest possible value, 31.0, as an approximation. Suppose further that this is an intermediate value in a longer computation. If you later need to compute half of the result, you will get 15.5, which is far from what the correct value would have been.

The same problem occurs in the underflow region. If you store 0.0 as an approximation of 0.021025, and you later want to multiply the value by 12.0, you will get 0.0. You risk being misled by what appears to be a reasonable value.

The problems encountered with overflow and underflow are alleviated somewhat by introducing more special values for the bit patterns. As is the case with zero, you must use some bit patterns that would otherwise be used to represent other values on the number line. In addition to zero, three special values are common—infinity, not a number (NaN), and denormalized numbers. The table below lists the four special values for floating-point representation and their bit patterns.

Special Value	Exponent	Significand
Zero	All zeros	All zeros
Denormalized	All zeros	Non-zero
Infinity	All ones	All zeros
Not a Number (NaN)	All ones	Non-zero

0.17.4.1 Infinity

Infinity is used for values that are in the overflow regions. If the result of an operation overflows, the bit pattern for infinity is stored. If further operations are done on this bit pattern, the result is what you would expect for an infinite value. For example, $\frac{3}{\infty} = 0$, $5 + \infty = \infty$, and the square root of infinity is infinity. You can produce infinity by dividing by 0. For example, $\frac{3}{0} = \infty$, and $\frac{-4}{0} = -\infty$. If you ever do a computation with real numbers and get infinity, you know that an overflow occurred somewhere in your intermediate results.

0.17.4.2 Not a Number (NaN)

A bit pattern for a value that is not a number is called a *NaN* (rhymes with *plan*). NaNs are used to indicate floating point operations that are illegal. For example, taking the square root of a negative number produces NaN, and so does dividing 0/0. Any floating point operation with at least one NaN operand produces NaN. For example, $7 + NaN = NaN$, and $\frac{7}{NaN} = NaN$.

Both infinity and NaN use the largest possible value of the exponent for their bit patterns. That is, the exponent field is all 1s. The significand is all 0s for infinity and can be any nonzero pattern for NaN. Reserving these bit patterns for infinity and NaN has the effect of reducing the range of values that can be stored. For a three-bit exponent and four-bit significand, the bit patterns for the largest magnitudes and their decimal values are

$1\ 111\ 0000\ (\text{bin}) = -\infty$

$1\ 110\ 1111\ (\text{bin}) = -15.5$

$0\ 110\ 1111\ (\text{bin}) = +15.5$

$0\ 111\ 0000\ (\text{bin}) = +\infty$

0.17.4.3 Denormalized Numbers

There is no single infinitesimal value for the underflow region in Fig 0.32 that corresponds to the infinite value in the overflow region. Instead, there is a set of values called *denormalized values* that alleviate the problem of underflow. Fig 0.33 is a drawing to scale of the floating point values for a binary representation without denormalized special values (top) and with denormalized values (bottom) for a system with a three-bit exponent and four-bit significand. The figure shows three complete sequences of values for exponent fields of 000, 001, and 010 (excess-3), which represent -3, -2, and -1 (dec), respectively.

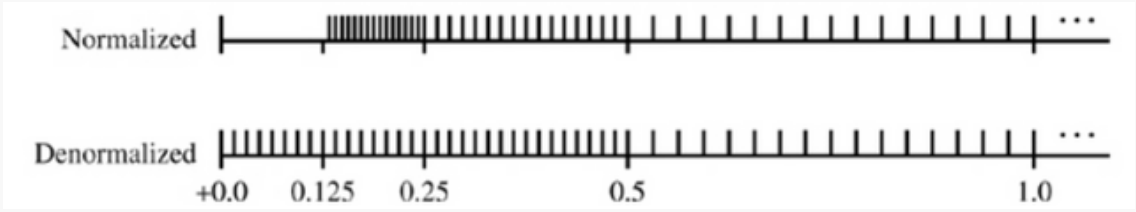


Fig 0.33. The real number line with and without denormalized numbers

For normalized numbers in general, the gap between successive values doubles with each unit increase of the exponent. For example, in the number line on the top, the group of 16 values between 0.125 and 0.25 corresponds to numbers written in binary scientific notation with a multiplier of 2^{-3}. The 16 numbers between 0.25 and 0.5 are spaced twice as far apart and correspond to numbers written in binary scientific notation with a multiplier of 2^{-2}.

Without denormalized special values, the gap between $+0.0$ and the smallest positive value is excessive compared to the gaps in the smallest sequence. Denormalized special values make the gap between successive values for the first sequence equal to the gap between successive values for the second sequence. It spreads these values out evenly as they approach $+0.0$ from the right. On the left half of the number line, not shown in the figure, the negative values are spread out evenly as they approach -0.0 from the left.

0.17.4.4 Gradual Underflow

This behavior of denormalized values is called gradual underflow. With gradual underflow, the gap between the smallest positive value and zero is reduced considerably. The idea is to take the nonzero values that would be stored with an exponent field of all 0s (in excess representation) and distribute them evenly in the underflow gap.

Because the exponent field of all 0s is reserved for denormalized numbers, the smallest positive normalized number becomes $0\ 001\ 0000 = 1.000 \times 2^{-2}$ (bin) = 0.25 (dec)

It might appear that we have made matters worse because the smallest positive normalized number with 000 in the exponent field is 0.1328125. But, the denormalized values are spread throughout the gap in such a way as to actually reduce it.

When the exponent field is all 0s and the significand contains at least one 1, special rules apply to the representation. Assuming a three-bit exponent and a four-bit significand, the rules are:

1. The hidden bit to the left of the binary point is assumed to be 0 instead of 1.
2. The exponent is assumed to be stored in excess-2 instead of excess-3.

For a representation with a three-bit exponent and four-bit significand, what decimal value is represented by $0\ 000\ 0110$? Because the exponent is all 0s and the significand contains at least one 1, the number is denormalized. Its exponent is 000 (excess-2) $= 0 - 2 = -2$ (dec), and its hidden bit is 0, so its binary scientific notation is 0.0110×2^{-2}. The exponent is in excess-2 instead of excess-3 because this is the special case of a denormalized number. Converting to decimal yields 0.09375.

To see how much better the underflow gap is, compute the values having the smallest possible magnitudes, which are denormalized.

$1\ 000\ 0001\ (\text{bin}) = -0.015625$

$1\ 000\ 0000\ (\text{bin}) = -0.0$

$0\ 000\ 0000\ (\text{bin}) = +0.0$

$0\ 000\ 0001\ (\text{bin}) = +0.015625$

Without denormalized numbers, the smallest positive number is 0.1328125, so the gap has been reduced considerably.

Fig 0.34 shows some of the key values for a three-bit exponent and a four-bit significand using all four special values. The values are listed in numeric order from smallest to largest. The figure shows why an excess representation is common for floating point exponents. Consider all the positive numbers from $+0.0$ to $+\infty$, ignoring the sign bit. You can see that if you treat the rightmost seven bits to be a simple unsigned integer, the successive values increase by one all the way from $000\ 0000$ for 0 (dec) to $111\ 0000$ for ∞. To do a comparison of two positive floating point values, say in a conditional statement like `if (x < y)` the computer does not need to extract the exponent field or insert the hidden bit. It can simply compare the rightmost seven bits as if they represented an integer to determine which floating point value has the larger magnitude. The circuitry for integer operations is considerably faster than that for floating point operations, so using an excess representation for the exponent really improves performance.

	Binary	Scientific Notation	Decimal
Not a number	1 111 nonzero		
Negative infinity	1 111 0000		$-\infty$
Negative normalized	1 110 1111	-1.1111×2^{3}	−15.5
	1 110 1110	-1.1110×2^{3}	−15.0
	...	...	...
	1 011 0001	-1.0001×2^{0}	−1.0625
	1 011 0000	-1.0000×2^{0}	−1.0
	1 010 1111	-1.1111×2^{-1}	−0.96875
	...	...	...
	1 001 0001	-1.0001×2^{-2}	−0.265625
	1 001 0000	-1.0000×2^{-2}	−0.25
Negative denormalized	1 000 1111	-0.1111×2^{-2}	−0.234375
	1 000 1110	-0.1110×2^{-2}	−0.21875
	...	...	...
	1 000 0010	-0.0010×2^{-2}	−0.03125
	1 000 0001	-0.0001×2^{-2}	−0.015625
Negative zero	1 000 0000		−0.0
Positive zero	0 000 0000		+0.0
Positive denormalized	0 000 0001	0.0001×2^{-2}	0.015625
	0 000 0010	0.0010×2^{-2}	0.03125
	...	...	...
	0 000 1110	0.1110×2^{-2}	0.21875
	0 000 1111	0.1111×2^{-2}	0.234375
Positive normalized	0 001 0000	1.0000×2^{-2}	0.25
	0 001 0001	1.0001×2^{-2}	0.265625
	...	...	...
	0 010 1111	1.1111×2^{-1}	0.96875
	0 011 0000	1.0000×2^{0}	1.0
	0 011 0001	1.0001×2^{0}	1.0625
	...	...	...
	0 110 1110	1.1110×2^{3}	15.0
	0 110 1111	1.1111×2^{3}	15.5
Positive infinity	0 111 0000		$+\infty$
Not a number	0 111 nonzero		

Fig 0.34. Floating-point values for a three-bit exponent and four-bit significand.

The same pattern occurs for the negative numbers. The rightmost seven bits can be treated like an unsigned integer to compare magnitudes of the negative quantities. Floating-point quantities would not have this property if the exponents were stored using two's complement representation.

Fig 0.34 shows that -0.0 and $+0.0$ are distinct. At this low level of abstraction, negative zero is stored differently from positive zero. However, programmers at a higher level of abstraction expect the set of real number values to have only one zero, which is neither positive nor negative. For example, if the value of `x` has been computed as -0.0 and `y` as $+0.0$, then the programmer should expect `x` to have the value 0 and `y` to have the value 0, and the expression `(x < y)` to be false. Computers must be programmed to return false in this special case, even though the bit patterns indicate that `x` is negative and `y` is positive. The system hides the fact that there are two representations of zero at a low level of abstraction from the programmer at a higher level of abstraction.

With denormalization, to convert from decimal to binary you must first check if a decimal value is in the denormalized range to determine its representation. From Figure 0.34, for a three-bit exponent and a four-bit significand, the smallest positive normalized value is 0.25. Any value less than 0.25 is stored with the denormalized format.

For a representation with a three-bit exponent and four-bit significand, how is the decimal value –0.078 stored? Because 0.078 is less than 0.25, the representation is denormalized, the exponent is all zeros, and the hidden bit is 0. Converting to binary, 0.078 (dec) = $0.000100111\ldots$. Because the exponent is all zeros and the exponent is stored in excess-2 representation, the multiplier must be 2^{-2}. In binary scientific notation with a multiplier of 2^{-2}, $0.000100111\ldots = 0.0100111\ldots \times 2^{-2}$. As expected, the digit to the left of the binary point is 0, which is the hidden bit. The bits to be stored in the significand are the first four bits of $.0100111\ldots$, which rounds off to $.0101$. So the floating point representation for -0.078 is $1000\ 0101$.

0.17.5 The IEEE 754 Floating-Point Standard

The Institute of Electrical and Electronic Engineers, Inc. (IEEE), is a professional society supported by its members that provides services in various engineering fields, one of which is computer engineering. The society has various groups that propose standards for the industry. Before the IEEE proposed its standard for floating point numbers, every computer manufacturer designed its own representation for floating point values, and they all differed from each other. In the early days before networks became prevalent and little data was shared between computers, this arrangement was tolerated.

Even without the widespread sharing of data, however, the lack of a standard hindered research and development in numerical computations. It was possible for two identical programs to run on two separate machines with the same input and produce different results because of the different approximations of the representations.

The IEEE set up a committee to propose a floating point standard, which it did in 1985. There are two standards: number 854, which is more applicable to handheld calculators than to other computing devices, and number 754, which was widely adopted for computers. The standard was revised with little change in 2008. Virtually every computer manufacturer now provides floating point numbers for their computers that conform to the IEEE 754 standard.

The floating point representation described earlier in this section is based on the IEEE 754 standard with the exception that the number of bits in the exponent field and in the significand are different as it will be difficult to explain. Fig 0.35 shows the two formats for the standard. The single-precision format has an 8-bit cell for the exponent using excess-127 representation (except for denormalized numbers, which use excess-126) and 23 bits for the significand. It corresponds to C-type float. The double-precision format has an 11-bit cell for the exponent using excess-1023 representation (except for denormalized numbers, which use excess-1022) and a 52-bit cell for the significand. It corresponds to C-type double.

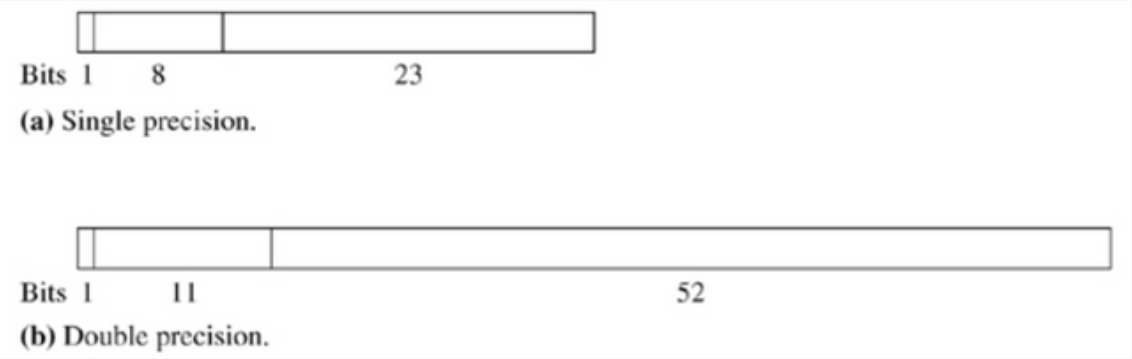


Fig 0.35. The IEEE 754 floating-point standard

The single-precision format has the following bit values. Positive infinity is

$0\ 1111\ 1111\ 000\ 0000\ 0000\ 0000\ 0000\ 0000$

The hexadecimal abbreviation for the full 32-bit pattern arranges the bits into groups of four as

$0111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000$

which is written $7F80\ 0000$ (hex). The largest positive value is

$0\ 1111\ 1110\ 111\ 1111\ 1111\ 1111\ 1111\ 1111$

or $7F7F\ FFFF$ (hex). It is exactly $2^{128}-2^{104}$, which is approximately 2^{128} or 3.4×10^{38}. The smallest positive normalized number is

$0\ 0000\ 0001\ 000\ 0000\ 0000\ 0000\ 0000\ 0000$

or $0000\ 0001$ (hex). It is exactly 2^{-149}, which is approximately 1.4×10^{-45}.

0.18 References

Hyde, R. (2012). *Write Great Code, Volume 1*. O'Reilly Media, Inc.

Kahan, W. (1997). *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic*. University of California.

Linda, N. (2018). *Essentials of computer organization and architecture* (Fifth edition.). Jones & Bartlett Learning, LLC.

Dale, N. (2016). *Computer science illuminated* (Sixth edition). Jones & Bartlett Learning, LLC.

Warford, J. S. (2017). *Computer systems* (Fifth edition). O'Reilly Media, Inc.