**Binary Representation of Numeric Values**

Lynn Robert Carter, Ph.D.

**Introduction**

People are familiar with the concept of integer and real values from their mathematics education. So it is common for people to assume that the similar computer concepts (integers and doubles in the programming language Java) would behave the same way. In many ways they are similar, but they are not exactly the same.

**Limitation on computer values**

One of the first major differences between mathematic values and computer values is the limitation on the range of values that can be expressed. When we think about the set of all integer values, it is common for us to believe that this set is infinite. By that, we mean that there is no largest or smallest value.

A common proof that the integers have no largest value uses contradiction. Let’s assume that there is indeed some largest integer value. Let’s call it *max*. Since max is the largest possible integer value, then it is not possible to compute the value *max* +1. We know this is not true, for we can always add one to an integer and come up with another valid integer. Therefore, the only thing we can conclude is that there is no largest possible value for the integers.

Within a computer, however, things are different. It is common for computer programs to use a fixed amount of storage to hold values that appear to be integers. For many applications, these computer representations do, indeed, operate just like the normal mathematical integers. There are times, however, where this similarity fails. Since computer values are stored in a fixed amount of computer memory, there is, indeed, a largest possible value that can be store in this fixed portion of memory.

**Binary values**

To understand the limit on the computer representation of integers, we need to know more about how the computer actually stores the data. Over the time that computers have existed, data has been stored in many different ways. Therefore, it is important to know some facts about the computer you are using. In the case of most computers that consumers might purchase today, values are stored in fixed sized blocks. The smallest of these blocks of memory is called a byte, and it is composed of eight binary bits, each bit can be either *on* or *off*.

Using combinatorics, we can determine the number of unique values that can be stored in an eight-bit byte using the following formula:

Number of possible combinations of eight bits is: 28 , which is 256.

These 256 values look like “00000000”, “00000001”, “00000010”, “00000011”… all the way up to “11111110”, “11111111”.

How we assign mathematical values to these patterns is up to us, just as those in the west and the east came up with different representations of the numeric value of one and two. By convention, we consider the bit pattern “00000000” to represent the value zero and the bit pattern “00000001” to be the value one.

Now we must decide if we are going to use all of these eight bits to hold just positive values (unsigned) or positive and negative values (signed). Clearly, choosing to express both positive and negative values reduces the total number of values that can be expressed. For the purpose of this paper, we choose signed values, with roughly half the values being positive and half being negative.[[1]](#footnote-1)

To implement signed values, the common convention today is to refer to the left-most bit as the “sign bit”. When this bit is “on”, the number is negative. When the bit is “off”, the values are “non-negative”.[[2]](#footnote-2) We will cover negative values later, so for now, we will assume that all values will have an “off” sign bit.

To make it easy to build computers, hardware designers treat the set of binary bits as if they were a base two number. Therefore, there is an easy mathematical way to translate the pattern of off and on bits into a normal mathematical integer and back.

In base ten numbers, each decimal digit can be one of ten different values. To represent this, we use ten unique characters. The value of each digit grows by a power of ten working to the left:

134 = 1 × 100 + 3 × 10 + 4 × 1

= 1 × 102 + 3 × 101 + 4 × 100

With binary values, the same is true, but instead if the value growing by a power of ten, working to the left, the value grows by a power of two. (We are only considering seven bit values, since we are assuming just positive value and we are ignoring the sign bit.) Therefore the “meaning” of the following bit pattern can be computed as follows:

0000101 = 0 × 1000000 + 0 × 100000 + 0 × 10000 + 0 × 1000 + 1 × 100 + 0 × 10 + 1 × 1

= 0 × 26 + 0 × 25 + 0 × 24 + 0 × 23 + 1 × 22 + 0 × 21 + 1 × 20

= 0 + 0 + 0 + 0 + 4 + 0 + 1

= 5

Using this approach, we can determine the largest possible positive value that can be represented in a signed byte value. The largest value would be one where all seven[[3]](#footnote-3) of the bits would be “on”, so the following show how we can determine what that the decimal equivalent value might be:

1111111 = 1 × 1000000 + 1 × 100000 + 1 × 10000 + 1 × 1000 + 1 × 100 + 1 × 10 + 1 × 1

= 1 × 26 + 1 × 25 + 1 × 24 + 1 × 23 + 1 × 22 + 1 × 21 + 1 × 20

= 64 + 32 + 16 + 8 + 4 + 2 + 1

= 127

**Negative binary values**

There are several schemes for processing negative values. The “two’s complement” method is the only one that has survived, because it was easier to produce the circuits that could perform addition and subtraction.

To convert a binary value from positive to negative, two steps are required:

1. Invert the bits (change each “on” to “off” and each “off” to “on”)
2. Add one (propagating the carries as required)

This same process is used to convert negative numbers to positive.

Given the eight-bit positive value “00001011”, convert it to a negative value using two’s complement, and then convert it back:

Original value: 00001011

Invert the bits: 11110100

Add one: 00000001

Negative value: 11110101

Negative value: 11110101

Invert the bits: 00001010

Add one: 00000001

Positive value: 00001011

**Converting Base Ten to Binary Values**

The process of computing the binary equivalent of a base ten number is easy to do. We will provide you two different methods, each works properly, so you can choose which-ever one is the easiest for you to use.

The first method is the “divide by two” method. The algorithm requires you to divide the number repeatedly by two and recording a zero bit for each time the division produced no remainder, and a one bit each time the division produced a remainder. The following shows the algorithm applied to the number 167:

Value = 107

Divide 107 by 2 = 53 plus remainder of 1

Add the remainder to the result: 1

Divide 53 by 2 = 26 plus a remainder of 1

Add the remainder to the result: 11

Divide 26 by 2 = 13 plus no remainder

Add a zero to the result: 011

Divide 13 by 2 = 6 plus a remainder of 1

Add the remainder to the result: 1011

Divide 6 by 2 = 3 plus no remainder

Add a zero to the result: 01011

Divide 3 by 2 = 1 plus a remainder of 1

Add the remainder to the result: 101011

Divide 1 by 2 = 0 plus a remainder of 1

Add the remainder to the result: 1101011

The “left to right” method requires you to write out the powers of two up to the first power of two that is larger than the value you wish to convert. In this method, you work left to right. Start with the largest power of two that is less than the value and make that the “current power of two”. Set the “remainder” to be the value to be converted.

If the current power of two is less than or equal to the remainder, add a one bit to the right side of the result and subtract the current power of two from the remainder, otherwise, add a zero bit to the right side of the result. Divide the power of two by two and do this again as long as the power of two is positive.

Remainder = 107

Current power of two = 64

Current power of two is less than or equal to the remainder, 107

Set the Remainder = 107 – 64 = 43

Add a one to the right side of the result: Result = 1

Divide the current power of two by two: Current power of two = 32

Current power of two is less than or equal to the remainder, 43

Set the Remainder = 43 – 32 = 11

Add a one to the right side of the result: Result = 11

Divide the current power of two by two: Current power of two = 16

Current power of two is not less than or equal to the remainder, 11

Add a zero to the right side of the result: Result = 110

Divide the current power of two by two: Current power of two = 8

Current power of two is less than or equal to the remainder, 11

Set the Remainder = 11 – 8 = 3

Add a one to the right side of the result: Result = 1101

Divide the current power of two by two: Current power of two = 4

Current power of two is not less than or equal to the remainder, 3

Add a zero to the right side of the result: Result = 11010

Divide the current power of two by two: Current power of two = 2

Current power of two is less than or equal to the remainder, 3

Set the Remainder = 3 – 2 = 1

Add a one to the right side of the result: Result = 110101

Divide the current power of two by two: Current power of two = 1

Current power of two is less than or equal to the remainder, 1

Set the Remainder = 1 – 2 = 1

Add a one to the right side of the result: Result = 1101011

Divide the current power of two by two: Current power of two = 0

The result is 1101011

**Working with larger values**

Integer values that can be held within a single byte are not always enough for most computer programs. The Java programming language provides programmers with integer values the can be one, two, four, and eight bytes in length. These integer types are called: byte, short, int, and long in the Java language.

We know that the largest possible positive value that can be stored in a byte is: 11111112  which is the value 12710. Most computer programs need to work with numbers that are larger than this. The next larger value is the “short” integer values, those that are two bytes in length (sixteen bits total, with one bit for the sign and 15 bits for the value).

The same methods for converting base two values to base ten works for these larger values with the only change being the number of bits involved. The following is a slightly more compact way to implement the “left to right” method to convert the number 41810 to base two:

418 = 1 × 28 + (418 – 256)

= 1 × 28 + 162

= 1 × 28 + 1 × 27 + (162-127)

= 1 × 28 + 1 × 27 + 35

= 1 × 28 + 1 × 27 + 0 × 26 + 35 26 is 64 > 35

= 1 × 28 + 1 × 27 + 0 × 26 + 1 × 25 + (35 – 32)

= 1 × 28 + 1 × 27 + 0 × 26 + 1 × 25 + 3

= 1 × 28 + 1 × 27 + 0 × 26 + 1 × 25 + 0 × 24 + 3 24 is 16 > 3

= 1 × 28 + 1 × 27 + 0 × 26 + 1 × 25 + 0 × 24 + 0 × 23 + 3 23 is 8 > 3

= 1 × 28 + 1 × 27 + 0 × 26 + 1 × 25 + 0 × 24 + 0 × 23 + 0 × 22 + 3 22 is 4 > 3

= 1 × 28 + 1 × 27 + 0 × 26 + 1 × 25 + 0 × 24 + 0 × 23 + 0 × 22 + 1 × 21 + (3 – 2)

= 1 × 28 + 1 × 27 + 0 × 26 + 1 × 25 + 0 × 24 + 0 × 23 + 0 × 22 + 1 × 21 + 1

= 1 × 28 + 1 × 27 + 0 × 26 + 1 × 25 + 0 × 24 + 0 × 23 + 0 × 22 + 1 × 21 + 1 × 20

= 1 1 0 1 0 0 0 1 1

= 110100011

This value requires nine bits to represent the value, so it could not be stored in a byte integer variable. It could be stored in all of the others (short, int, and long). If the value were to be stored in a short int variable, the bit pattern would be a zero bit at the left for the sign bit (the value is positive), six zero bits, and then the nine bits from above:

= 0000000110100011

Using binary with values larger than byte integers is tedious and error prone. It is for this reason that we represent binary value in either base 8 or base 16. Unlike the conversion from base 2 to base 10, you do not need to compute the conversion. You only need to group the bits together and look up values in a table.

**Hexadecimal and Octal Representations**

It is common to use Hexadecimal, often just called “hex” even though that could be misunderstood to be six, to represent binary values. (Everyone knows that base 6 would make no sense, so this nickname does not cause problem.) Some computer makes prefer to use octal, base 8, but hexadecimal is much more common.

If one is given a base two number, converting it to hexadecimal is simple. Start by grouping the value in block of four binary bits.

0000 0001 1010 0011

Each of these groups can now be represented by one of sixteen different hexadecimal digits. The

normal decimal digits are used for the first ten digits and the letters “A” through “F” are used for the last six.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0000 | 0 | 0100 | 4 | 1000 | 8 | 1100 | C |
| 0001 | 1 | 0101 | 5 | 1001 | 9 | 1101 | D |
| 0010 | 2 | 0110 | 6 | 1010 | A | 1110 | E |
| 0011 | 3 | 0111 | 7 | 1011 | B | 1111 | F |

For each block of four binary bits, find it in the table and then use the hexadecimal equivalent to the right as shown here:

0000 0001 1010 0011

0 1 A 3

As you can see, these four characters are much easier to get right that a string of sixteen zeros and ones. When one works with int (32 bits or 8 hexadecimal characters) or long integers (64 bits or 16 hexadecimal characters), the benefits grow.

Using octal is similar. Instead of groups of four bits, octal requires you to group the bits in block of three. Since sixteen bits does not divide evenly by three, for any of these values, the left hand character is restricted to just one or two bits. For example, in the sample from above, the left-most group is just a single bit:

0 000 000 110 100 011

0 0 0 6 4 3

**Representing Real Values**

This section will only be a very brief introduction to the use of binary to represent real values. Your textbook, Appendix E, covers how Java represents reals more completely.

As with integers, real values are infinite, only they are even more so. The real numbers do not have a largest or smallest value. They also accurately represent things that are difficult for us to represent with characters. For example, the notion of one third is easy to express in English. We run into problems representing this simple value in base two and in base ten.

The starting point for representing real values is to consider these values of being composed to two parts, the whole part and the fractional part. If one needed to represent the value 12.75, we can think of this as the whole value to the left of the decimal point and the fractional value to the right of the decimal point.

Left of the decimal point = b6 × 26 + b5 × 25 + b4 × 24 + b3 × 23 + b2 × 22 + b1 × 21 + b0 × 20

= 0 × 64 + 0 × 32 + 0 × 16 + 1 × 8 + 1 × 4 + 0 × 2 + 0 × 1

= 0 0 0 1 1 0 0

Right of the decimal point = f1 × 2-1 + f2 × 2-2 + f3 × 2-3 + f4 × 2-4 + f5 × 2-5 + f6 × 2-6

= 1 × .5 + 1 × .25 + 0 × .125 + 0 × .0625 + 0 × .03125 + 0 × .015625

= 1 1 0 0 0 0

0001100.110000 in base two.

In this example, the powers of two work down from six to zero on the left side of the decimal point, representing the whole value portion of the value. They then continue into negative values on the right side, representing the fractional powers of two. A representation of a real value is then a sum of these whole and fractional powers of two.

As we mentioned earlier, this representation can be very close to many numbers without being precise. In the case of 12.75, the binary representation here is precise. This is not the case for many numbers, where a repeating pattern of ones and zeros continues to infinity, just as the sequence of “3” digits continues to the right to infinity when one tries to represent the fraction one third.

**Floating Point Values**

In Java, the computer is required to represent floats and doubles using a specified floating-point format known as “IEEE 754”. The very simple representation provided above introduces a basic concept, but the scientific computation needs numbers that are very large and very small. Using the representation introduced above would require far too much storage and would cause the computer to run far more slowly than would be desirable.

The heart of the IEEE 754 representation is notion of scientific notation. Scientists and those doing computation with real values are used to work with number in scientific notation. Here, the number is divided into two different parts, the mantissa, a value that is usually less than ten and greater than one tenth, and an exponent. A typical example of a number is scientific notation would be:

6.022 × 1023

Using number of this form makes it easy for a very large range of numbers to be represented as well as numbers that approach very close to zero, precisely the kinds of numbers that scientists need to use. The Appendix E does a nice introduction to IEEE 754 format numbers, if you want to learn more about them.

1. In fact, there is precisely one more negative number than positive number. There is one more negative number due to the fact that zero was removed from the positive values. [↑](#footnote-ref-1)
2. Notice the use of “non-negative” as opposed to “positive”. The value of zero is non-negative, but it is not positive. Some old computers hand the concept of positive and negative zero, but those machines are not widely used, if they are even functioning today. [↑](#footnote-ref-2)
3. Remember, the sign bit for positive value must be zero, so only seven bits are used to express the positive or negative value. [↑](#footnote-ref-3)