Representing Integers in a Computer
(Revised 12 April 2004)

The basic strategy for representing integers in a computer is the binary number system. But the details of the representation are affected by two parameters: the number of bits allocated to storing the integer, and the method of dealing with negative numbers. If N bits are allocated, it is possible to store representations of up to 2^N different integers. For example, with 8 bits, only 256 integers can be represented, but with 32 bits, 4,294,967,296 integers can be represented. In some applications, negative numbers are not needed, and integers can be considered as unsigned. But in most applications, it is necessary to work with negative integers as well as zero and positive integers. There are four different methods commonly used for dealing with negative numbers: sign-magnitude, one's complement, two's complement, and excess-M. Each of these will be discussed below.

In all of the examples in this section, we will assume that 12 bits are allocated to store an integer. This will permit us to illustrate adequately all of the principles without dealing with large numbers.

Unsigned representation

The simplest way of using N bits to represent integers is to use the binary number system directly. Then it is possible to represent the integers from 0 to 2^N-1. For example, with 12 bits we can represent the numbers from 0 to 4095. `000000000000` represents 0, `000000000001` represents 1, and `111111111111` represents 4095. This method of representing integers is called *unsigned*.

But we encounter a problem as soon as we try to do arithmetic. Let's start with addition. The sum of two numbers in the range 0-4095 might not be in the range 0-4095. For example, 3210 + 2012 = 5222, which cannot be represented in this system. So it is impossible for any computer using the unsigned method to do all addition problems correctly. In fact, almost half of all possible addition problems must be done incorrectly!

The condition in which the sum of two numbers is outside the range of numbers that can be represented is called *overflow*. Overflow is an inherent characteristic of computer arithmetic; it may take any of several forms, but it cannot be eliminated.

Unsigned arithmetic is in wide use in situations in which negative numbers do not occur. In particular, arithmetic with memory addresses is unsigned arithmetic.

Sign-magnitude representation

The sign-magnitude method is the simplest way of handling negative numbers - at least from a human the point view. The idea is to set aside one bit to store the sign of the integer (either + or -) and use the remaining *N*-1 bits to store the magnitude of the number. With 12 bits, this method allows representing integers from $-(2^{11}-1)$ to $+(2^{11}-1)$, that is, from -2047 to +2047. By convention, the left-most bit is the sign bit, and a sign bit of 1 signifies a negative number. So in a 12-bit sign-magnitude system,

`111111111111`	represents -2047
`100000000001`	represents -1
`100000000000`	represents 0 (so-called negative 0)
`000000000000`	represents 0 (so-called positive 0)
`000000000001`	represents +1
`011111111111`	represents +2047

Note that there are two different representations for the number zero! These representations are called positive zero and negative zero, but it is important to remember that they represent the same number (they are mathematically equal) and that in mathematics, zero is neither positive nor negative.

The problem of overflow persists. For example, 1400 + 1500 cannot be done correctly. Neither can (-1400) + (-1500).

The sign-magnitude method is convenient for human users of data, but it presents a more difficult problem for the electrical engineers who must design the electronic circuits to do arithmetic computations. To add two numbers, two cases must be distinguished: the summands are of the same sign or of different signs. In the first case, the magnitudes must be added, and the result given the sign of the summands. In the second case, the smaller magnitude must be subtracted from the larger magnitude, and the result given the sign of the summand having the larger magnitude. So the engineer must incorporate in his circuit both an adder and a subtractor, and circuits to distinguish the various cases. The process of computing the sum of two numbers in sign-magnitude arithmetic is shown in Figure 2-1. The process is, of course, exactly the method taught in school for adding signed numbers by hand.

Two's complement representation

A two's complement adder is an electronic circuit that accepts two bit strings as input and does a simple binary addition. If the addition results in a *carry* from the leftmost position, it is simply discarded. Such a circuit is much simpler than a sign-magnitude adder, since it does not distinguish between two cases. Because it is simpler, it is (all other parameters being equal) faster.

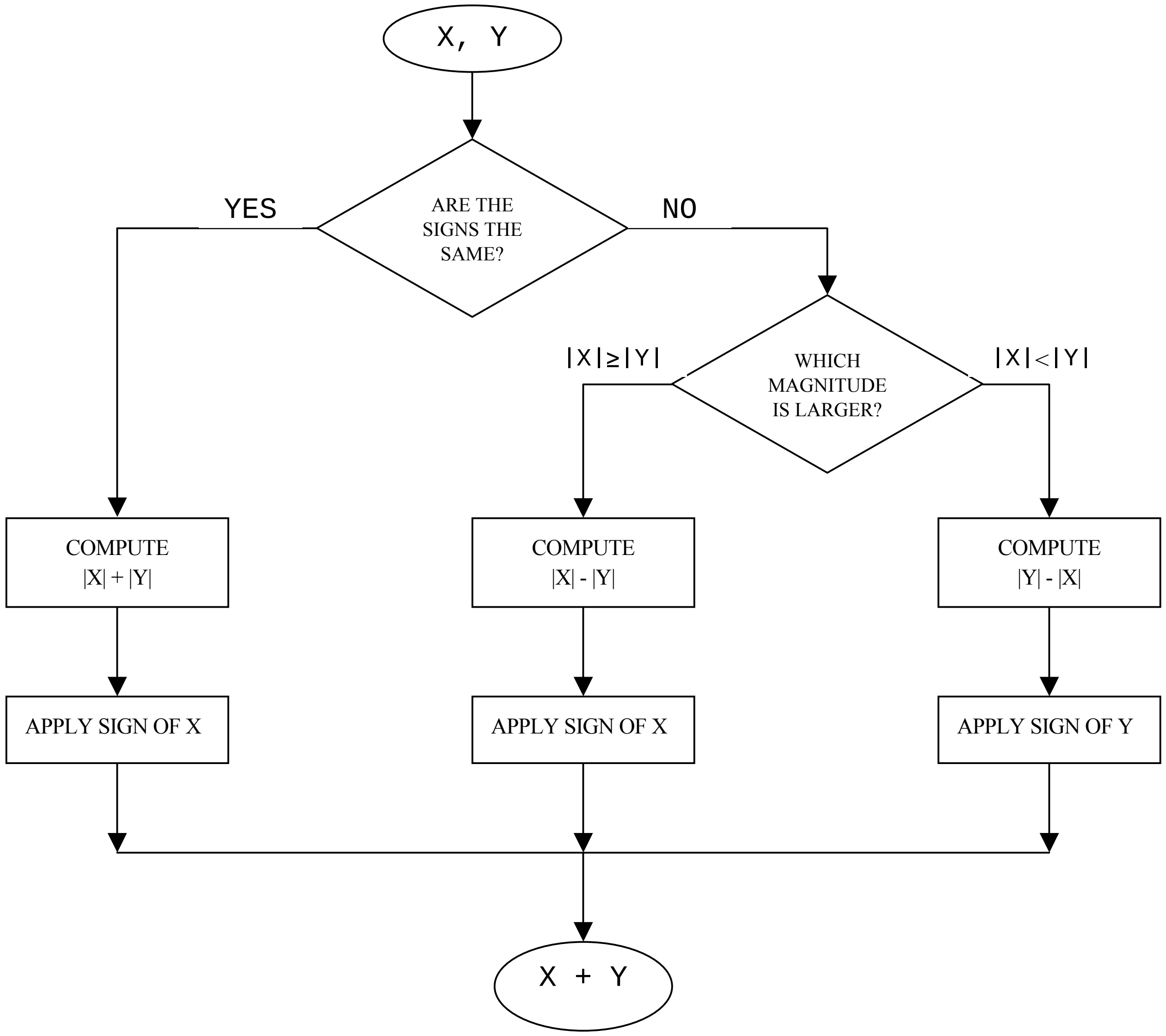

Figure 2-1. Sign-magnitude addition

Let's look at what a two's complement adder does to several pairs of 12-bit integers.

Example 1. Add 23 and 1018.

```
    0000 0001 0111  (This represents 23)
 +  0011 1111 1010  (This represents 1018)
    --------------
    0100 0001 0001  (This represents 1041)
```

If we interpret these numbers the same way we interpret unsigned and sign-magnitude numbers, this addition problem is done correctly.

Example 2. Determine what number is represented by `1111 1111 1111`. We will choose a number - any number will do - and add `1111 1111 1111` to it.

```
    0000 0001 0111  (This represents 23)
 +  1111 1111 1111  (What does this represent?)
    --------------
    0000 0001 0110  (This represents 22)
```

To understand the meaning of `1111 1111 1111`, we have to solve the equation

$$23 + x = 22$$

Of course, the solution is $x = -1$. It follows that the bit pattern `1111 1111 1111` represents -1. A similar exercise shows that `1111 1111 1110` represents -2.

Example 3. Determine what number is represented by `1000 0000 0000`. We can get `1111 1111 1111` from this number by adding `0111 1111 1111` to it.

```
    0111 1111 1111  (This represents 2047)
 +  1000 0000 0000  (What does this represent?)
    --------------
    1111 1111 1111  (This represents -1)
```

To understand the meaning of `1000 0000 0000`, we have to solve the equation

$$2047 + x = -1$$

The solution is $x = -2048$. It follows that the bit pattern `1000 0000 0000` represents -2048. ▌

In two's complement arithmetic, a bit string with a 0 in the most significant bit represents zero or a positive integer, while a bit strings with a 1 in the most significant position represents a negative integer. If N bits are allocated to storing an integer, then the method is capable of representing all integers in the range from -2^{N-1} to $+2^{N-1}-1$.

In the two's complement system, there is a simple rule for finding the representation for the negative of an integer: take the bitwise complement of the representation of the number and add 1. The bitwise complement of a bit string is formed by changing all 1 bits to 0 and all 0 bits to 1. For example, the bitwise complement of `0011 0101 1110` is `1100 1010 0001`. Let's look at two examples:

Example 4. The string `0011 0101 1110` represents 857. The bitwise complement is `1100 1010 0001`. Adding 1 to this gives `1100 1010 0010`, which represents -857. To verify this, add the two bit strings, remembering to discard the carry from the left end:

```
   0011 0101 1110  (This represents 857)
+  1100 1010 0010  (This represents -857)
   ______________
   0000 0000 0000  (This represents zero)
```

Example 5. The bit string `1111 1001 0011` represents some negative integer in the two's complement system. What integer? To answer this question, we first find its negative. The bitwise complement of `1111 1001 0011` is `0000 0110 1100`. Adding 1 gives `0000 0110 1101`, which represents 109. So `1111 1001 0011` represents -109.

It is easy to see why this method works in general. Let x be a bit string and let x' be its bitwise complement. Then all of the bits in $x + x'$ are 1's, so

$$x + x' = -1$$

Rearranging terms in this equation yields

$$-x = x' + 1$$

Overflow in two's complement arithmetic can occur in one of two ways: the sum of two positive integers may be larger than the largest positive integer that can be represented; or the sum of two negative integers may be smaller than the smallest negative integer represented. Here are examples of each.

Example 6. Add 1952 and 529 in 12-bit two's complement arithmetic.

```
    0111 1010 0000  (This represents +1952)
+   0010 0001 0001  (This represents +529)
    --------------
    1001 1010 0001  (This represents -1615 - a wrong answer)
```

Example 7. Add -1342 and -1703 in 12-bit two's complement arithmetic.

```
    1010 1100 0010  (This represents -1342)
+   1001 0101 1001  (This represents -1703)
    --------------
    0100 0001 1011  (This represents +1051 - a wrong answer)
```

One's complement representation

A one's complement adder is another electronic circuit that accepts two bit strings as input. It does a simple binary addition, as a two's complement adder does. However, if there is a carry from the leftmost position, a one's complement adder brings the carry over to the rightmost (units) position and continues to add. Here is an example:

Example 8.

```
    1110 1100 0010
+   1101 0101 1001
    --------------
    1100 0001 1011
+                1  (Here is the carry from the leftmost position)
    --------------
    1100 0001 1100
```

Of course, this answer is different from the output produced by a two's complement adder from the same inputs.

Example 9. Determine what number is represented by 1111 1111 1111 in 12-bit one's complement arithmetic. We can start with any number and see what happens when we add 1111 1111 1111 to it.

```
    0000 1100 0010  (This represents 194)
+   1111 1111 1111  (What does this represent?)
    --------------
    0000 1100 0001
+                1  (Here is the carry from the leftmost position)
    --------------
    0000 1100 0010  (This is 194 again.)
```

It follows that in the one's complement system, the bit string `1111 1111 1111` represents zero.

Example 9 shows that in one's complement arithmetic, as in the case of sign-magnitude arithmetic, there are two representations of zero: `0000 0000 0000`, called "positive zero", and `1111 1111 1111`, called "negative zero". In one's complement arithmetic, the negative of a number is formed by taking the bitwise complement. This works because the sum of a number and its bitwise complement consists of all 1 bits, which represents zero. If N bits are allocated to storing an integer in a one's complement system, the integers that can be represented are those in the range $-(2^{N-1}-1)$ to $+(2^{N-1}-1)$.

Excess-M representation

In the excess-M method of representing integers, the meaning of a bit string is determined by first interpreting it as an unsigned number, then subtracting M. Conversely, to find the excess-M representation of a number, first add M and then represent the sum as an unsigned integer.

Example 10. In 12-bit excess-2048 arithmetic,

`1000 1100 1110` represents 206 (2254 - 2048)
`0011 0101 1010` represents -1190 (858 - 2048)

Example 11. In 12-bit excess-2047 arithmetic,

37 is represented by 37+2047, or `1000 0010 0100`
-9 is represented by -9+2047, or `0111 1111 0110`

In N-bit unsigned arithmetic, the numbers that can be represented are 0 to 2^N-1. So in the N-bit excess-M system, the numbers that can be represented are $-M$ to 2^N-1-M. Usually M is chosen to be either 2^{N-1} or $2^{N-1}-1$.

The excess-M representation has very limited applications. One is in electronic equipment: devices called analog-to-digital converters, which measure voltages, usually produce outputs in an excess-M format. The second principal application is in representing exponents in floating-point representation of real numbers.