Worksheet: Signed Numbers

Until now, we haven't considered the representation of negative numbers. Binary numbers that are always interpreted to have positive values are called ***unsigned numbers***. There are many cases where programs will require the ability to represent negative numbers. When a binary representation of a number includes the ability to represent both positive and negative numbers, the representation is called a ***signed number***.

Sign and Magnitude

When we work with positive and negative denary values, we simply place a plus (+) or minus (−) sign in front of the number to represent positive and negative, respectively. As we have discussed, digital computers store and manipulate data as zeros and ones. There are no other symbols, so we will need to represent the positive and negative with bit values.

Perhaps the most obvious way to represent a negative number in binary is to use the *most significant bit* represent a positive or negative sign, and have the remaining bits represent an *unsigned number* as we've been working with in previous lessons. This representation is called ***sign and magnitude***, and the positive sign is represented by a zero while the negative sign is represented by a one.

There are some drawbacks to using *sign and magnitude*. Let's examine one of these.

1. Write the *sign and magnitude* equivalent the denary numbers 21 and −21 into the appropriate row, then add the binary digits.

Bit value	− / +	64	32	16	8	4	2	1
carry over								
denary +20 in *sign and magnitude*								
denary −20 in *sign and magnitude*								
sum								
denary conversion								

As we can see, with sign and magnitude, we cannot simply add positive and negative numbers. Another drawback is that there are two representations of zero – a positive zero and a negative zero (for an 8-bit *sign and magnitude* number, these are 0000_0000 and 1000_0000, respectively).

One's Complement

You should remember the word ***complement*** from your study of set theory. As an example, if we consider the universe, $U = \{ 1, 2, 3, 4, 5 \}$ and we have set $A = \{ 2, 5 \}$, then the complement of set A is $A^c = \{ 1, 3, 4 \}$. In set theory, taking the complement inverts (or flips) the element that are in the set with those outside of the set.

For a binary digit, the *complement* of zero is one and the complement of one is zero. A positive *one's complement* number is the same as an unsigned number. A negative *one's complement* number is found by taking the complement of every bit of the number. With this knowledge, complete the table below.

denary +20 in binary	0	0	0	1	0	1	0	0
denary −20 in *one's complement*								
sum								

Although the sum of these may look incorrect, if we realize that taking the one's complement of 0000_0000 is actually 1111_1111, our answer is actually correct. The problem is just that we have two representations of zero: a positive zero (0000_0000) and a negative zero (1111_1111).

There are other reasons that one's complement is not as nice as the next representation, but since *one's complement* is not included in the curriculum, we will end our discussion here. Just know that there is a representation called *one's complement* and it is not used very much in practice.

Two's Complement

To lead into our discussion of two's complement, let's first look a bit at the possible binary values that we can use to represent a number in binary. To keep the range of numbers small, let's examine all the binary numbers that can be represented with three binary digits. Starting at zero and repeatedly adding one to the number, we come up with the series: 000, 001, 010, 011, 100, 101, 110, 111.

If we add 1 to 111, we get the value 1000. This value does not fit into the three bits, so there is ***overflow*** with the carry over lost, and the resultant value is a repeat of the bits 000. This is represented in *Figure 1: Cyclical Three-bit Sequence*.

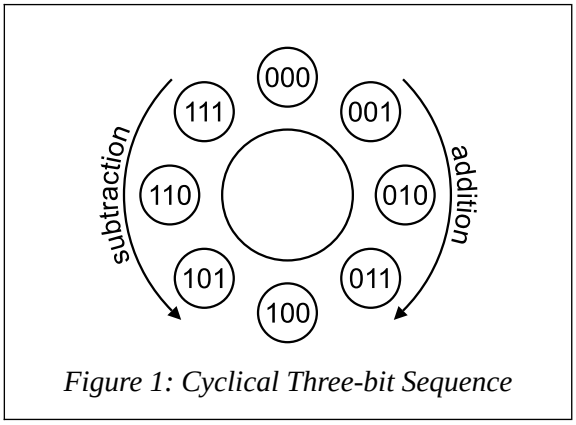

Figure 1: Cyclical Three-bit Sequence

Considering the addition of one to 111 results in a value of 000 suggests that we might want to represent negative one (−1) by the bits 111, negative two (−2) by the bits 110, and so on. This allows us to use the same algorithm for addition of positive values to both positive and negative numbers.

However, if we decide to use 111 to represent negative one (−1), it can no longer be used to represent positive seven (+7, the value that the binary digits would convert to if they were unsigned). There must be a dividing point for which values are positive and which are negative. A convenient point that almost exactly divides equally the number of positive and negative numbers is where the *most significant bit* changes from a zero to a one. For three bit numbers, all numbers from 001 to 011 represent positive numbers, and all numbers from 100 to 111 represent negative numbers.

This system of representing numbers is called ***two's complement***, and the denary equivalent of three-bit numbers represented using this system is given in *Figure 2: Two's Complement Representation*.

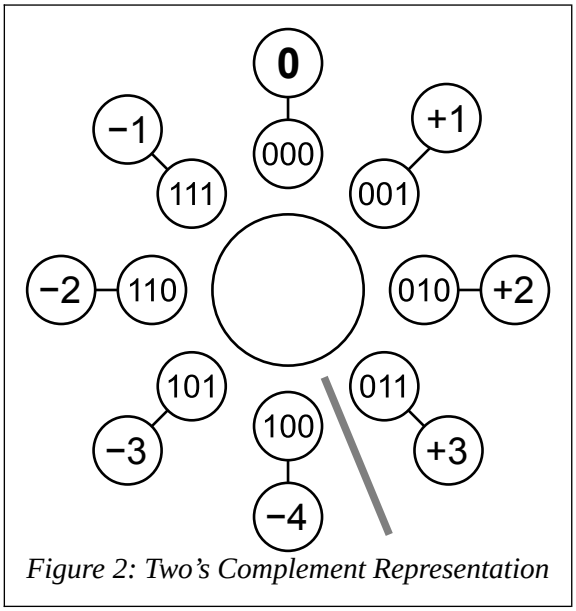

Figure 2: Two's Complement Representation

With two's complement, there is one greater negative number than positive number, there is only one representation of the value zero, and we can quickly tell if the number is positive or negative by looking only at the *most significant bit*.

Worksheet: Signed Numbers

Overflow With Two's Complement

With *two's complement*, an operation causes a carry over beyond the *most significant bit*, such as when adding 001 to 111 with the result of 000, this is not actually an ***overflow*** – it is just adding one to negative one, resulting in zero. An ***overflow*** occurs when an operation causes a carry over into the *most significant bit* with no carry over out of the *most significant bit*. For a three-bit two's complement number, this occurs, for example, when adding 001 to 011, resulting in 100. In that case, the maximum possible value represented (011, denary +3) is exceeded, resulting in the looping around to the value to the maximum negative possible value represented (100, denary −4).

Converting a Positive Binary Number to Negative Two's Complement

Converting a positive denary number into *two's complement* binary number is the same as we have done for every other representation (*unsigned*, *sign and magnitude*, and *one's complement*). To get the negative equivalent of the number in *two's complement*, the first step is the same as with one's complement – we take the complement (invert) each bit. The "two" from "*two's complement*" refers to needing a second step. For the second step, we add one to the result.

denary +20 in binary	0	0	0	1	0	1	0	0
denary −20 in *one's complement* (flip)								
denary −20 in *two's complement* (+1)								

Let's verify that we get zero when we take the sum.

carry over								
denary +20 in binary	0	0	0	1	0	1	0	0
denary −20 in *two's complement*								
sum								

When we add a positive number to its negative equivalent in *two's complement*, the result is equal to zero. With *two's complement*, there is exactly one representation of the value zero.

The thick-border table cells contain the carry-out (left) and carry-in (right) of the *most significant bit*. Since these are the same, there has been no overflow with the addition.

Converting Negative Two's Complement to a Positive Binary Number

So how to convert from a negative *two's complement* number back to a positive value? Interestingly (and thankfully easy to memorize), it's the same way as we converted to a negative value. I expect you'll need to see it work to believe it, so let's do it.

denary −20 in *two's complement*								
complement the bits (flip)								
add one (+1)								

Converting Directly From Two's Complement to Denary

The most obvious way to convert a negative *two's complement* number into denary is to first convert the number to the positive equivalent, then convert that number to a denary value in the same way as we have done in previous lessons.

The following table shows a method to convert directly from two's complement to denary. It is exactly the same as converting an unsigned number, but the leftmost bit value is negated.

Table 1: *Place Values for a Two's Complement Number*

bit position		7	6	5	4	3	2	1	0
place value	**exponential**	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	denary	−128	64	32	16	8	4	2	1
binary digits (bits)		1	1	0	1	0	1	1	0
denary conversion		-128 + 64 + 16 + 4 + 2 = -42							

Now try this for yourself.

2. Given the *two's complement* number 1010_0101, complete the table to convert to denary directly.

a)

bit position		7	6	5	4	3	2	1	0
place value	exponential								
	denary								
binary digits (bits)									
denary conversion (write the equation)									

b) Now verify this by converting to a positive number and then to denary.

two's complement	1	0	1	0	0	1	0	1
complement the bits (flip)								
add one (+1)								
denary conversion (write the equation)								

Summary

Using *two's complement* to represent *signed numbers* in binary has advantages over the alternatives we have examined. One advantage over *sign and magnitude* is that positive and negative numbers can be operated on in the same way when adding and subtracting numbers. Another advantage, over both *sign and magnitude* and *one's complement,* is that there is a single representation of the number zero. For these reasons, *two's complement* is the most common way to represent *signed numbers* in computers.