Two’s Complement for Programmers

There is a need to represent both positive and non-negative numbers in a binary format. The solution to this problem is a two’s complement representation where approximately half the binary numbers we use are negative, and the other half non-negative. When working with two’s complement integers, we have to decide how many bits will be devoted to each integer. The most important feature of two’s complement integers is that the leftmost bit represents the sign of the number. A zero (0) denotes a non-negative integer, and a one (1) denotes a negative integer. Determining the rest of the bits in an integer representation is more complicated. One thing to keep in mind is that two’s complement was designed to make life easier for electrical engineers designing circuits, not programmers reading a dump.

Let’s start by looking at two’s complement representations using 3 and 4 bits.

3-Bit, 2’s Complement

```
000 =  0
001 =  1
010 =  2
011 =  3
100 = -4
101 = -3
110 = -2
111 = -1
```

Notice that on the left side, we are simply listing the plain binary representations of the numbers from 0 to $2^3 - 1 = 7$. The right side integers are the the interpretation we give to those numbers. Further note that for the non-negative integers (0 - 3), the interpretation agrees with the plain binary representation.

When the high-order bit changes to 1, the interpretation changes. For example, we interpret 100 as -4. Continuing forward, the numbers become -3, -2, and -1. Let’s switch to a 4-bit two’s complement representation.

4-Bit, 2’s Complement

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

Notice that again on the left side, we are simply listing the plain binary representations of the numbers from 0 to $2^4 - 1= 15$. The right side integers are the the interpretation we give to those numbers. Further note that for the non-negative numbers (0 - 7), the interpretation agrees with the plain binary representation.

When the high-order bit changes to 1, the interpretation changes again. For example, we interpret 1000 as -8. Continuing forward, the numbers become -7, -6, …, -1. The interpretation we give to these binary numbers was chosen to make life easier for electrical engineers designing circuits, not programmers reading a dump. With these interpretations, it becomes easy to design adder circuits (and others).

As programmers we have to learn some techniques for dealing with these interpretations. Let me list a few facts that are helpful to remember.

- Two's Complement arithmetic is a way to use binary numbers to represent signed integers.
- Two's Complement was designed to make life easier for electrical engineers designing circuits, not programmers reading a dump.
- We choose to interpret some numbers as non-negative and some as negative.
- The high order bit is a sign (0 is + sign, 1 is - sign).
- With two's complement, you must specify how many bits you are using. 3-bit two's complement is completely different from 4-bit two's complement.
- Zero has a positive sign (0).
- A sequence of all zeroes is always zero, and a sequence of all 1's is -1 in every two's complement representation, no matter the number of bits.

There is one skill we need in order to deal with bit sequences that are negative : computing the additive complement.

Additive Complements

Since negative numbers are hard to interpret in 2's complement, we need a technique that produces the additive complement of the number we are dealing with. Here is the procedure.

1) Flip the bits. (Change all 1's to 0's and all 0's to 1's.)
2) Add 1.

Examples:

Here is a 4-bit two's complement integer: 1100 (The sign is negative.)
1) Flip the bits. 0011
2) Add 1 1
——
0100 = 4 in 4-bit two's complement.

Since 0100 is the additive complement of 1100, 1100 is -4

Here is another a 4-bit two's complement integer: 1001 (The sign is negative.)
1) Flip the bits. 0110
2) Add 1 1
——
0111 = 7 in 4-bit two's complement.

Since 0111 is the additive complement of 1001, 1001 is -7

Another Fact

Sign extending the leftmost bit to the left for any integer, changes the two's complement representation but preserves the value of the integer.

We just saw that 1001 is -7 in 4-bit two's complement. Extending the sign produces these results. Check the results yourself.

1001	= -7 in 4-bit two's complement
11001	= -7 in 5-bit two's complement
111001	= -7 in 6-bit two's complement
1111001	= -7 in 7-bit two's complement
11111001	= -7 in 8-bit two's complement

This fact also applies to positive integers.

0111	= 7 in 4-bit two's complement
00111	= 7 in 5-bit two's complement
000111	= 7 in 6-bit two's complement
0000111	= 7 in 7-bit two's complement

Adding Two’s Complement Integers

In working in 4-bit two’s complement, the range of numbers is -8 to +7. What happens if we compute -8 + -8, or 7 + 7. Obviously, the results can’t be represented in a 4-bit two’s complement representation since they are outside the range of representable values. The result is overflow or underflow. We will classify both situations as overflow. This can happen in any two’s complement representation. As a result, we need a way to detect overflow.

The way this is done is by looking at carries into and out of the leftmost (sign) bit. Here are the rules:

1)	Carry in and carry out of the sign bit	- the result is valid.
2)	No carry in and no carry out of the sign bit	- the result is valid
3)	Carry into the sign bit, no carry out	- the result is invalid (overflow).
4)	No carry into the sign bit, carry out of the sign bit	- the result is invalid (overflow).

Let’s look at four example additions in 4-bit two’s complement.

```
     1001   = -7
     1001   = -7
     ____
  1  0010   =  2      (No carry into the sign bit, carry out - invalid result)

     0111   =  7
     0111   =  7
     ____
     1110   = -2      (Carry into the sign bit, no carry out - invalid result)

     0011   =  3
     0011   =  3
     ____
     0110   =  6      (No carry into the sign bit, no carry out - valid result)

     1111   = -1
     1111   = -1
     ____
   1 1110   = -2      (Carry into the sign bit, Carry out - valid result)
```

Subtracting Two’s Complement Integers

Subtraction is accomplished by adding the additive complement. Here is an example in 4-bits. We want to compute (-7) - (-7) , but instead we add (-7) + (+7).

```
     1001   = -7
     0111   = +7
     ____
  1  0000   =  0      (Carry into the sign bit, carry out - valid result)
```

Using a Programmer’s Calculator

I urge you to learn to do the computations by hand before using an online programmer’s calculator. When you do choose an online calculator, be sure to pick one that allows you to specify the number of bits in the two’s complement representation - many don’t. Two’s complement integers are found in various sizes in assembler. Fullwords have 32 bits, halfwords have 16, many shift instructions use 6 bits, Y-type instructions use 20 bits. If your calculator only works with a fixed number of bits, it is generally useless.