Two's Complement

Thomas Finley, April 2000

Contents and Introduction

- Contents and Introduction
- Conversion from Two's Complement
- Conversion to Two's Complement
- Arithmetic with Two's Complement
- Why Inversion and Adding One Works

Two's complement is not a complicated scheme and is not well served by anything lengthly. Therefore, after this introduction, which explains what two's complement is and how to use it, there are mostly examples.

Two's complement is the way every computer I know of chooses to represent integers. To get the two's complement negative notation of an integer, you write out the number in binary. You then invert the digits, and add one to the result.

Suppose we're working with 8 bit quantities (for simplicity's sake) and suppose we want to find how -28 would be expressed in two's complement notation. First we write out 28 in binary form.

```
0 0 0 1 1 1 0 0
```

Then we invert the digits. 0 becomes 1, 1 becomes 0.

```
1 1 1 0 0 0 1 1
```

Then we add 1.

```
1 1 1 0 0 1 0 0
```

That is how one would write -28 in 8 bit binary.

Conversion from Two's Complement

Use the number 0xFFFFFFFF as an example. In binary, that is:

```
1111  1111  1111  1111  1111  1111  1111  1111
```

What can we say about this number? It's first (leftmost) bit is 1, which means that this represents a number that is negative. That's just the way that things are in two's complement: a leading 1 means the number is negative, a leading 0 means the number is 0 or positive.

To see what this number is a negative of, we reverse the sign of this number. But how to do that? The class notes say (on 3.17) that to reverse the sign you simply invert the bits (0 goes to 1, and 1 to 0) and add one to the resulting number.

The inversion of that binary number is, obviously:

```
0000  0000  0000  0000  0000  0000  0000  0000
```

Then we add one.

```
0000  0000  0000  0000  0000  0000  0000  0001
```

So the negative of 0xFFFFFFFF is 0x00000001, more commonly known as 1. So 0xFFFFFFFF is -1.

Conversion to Two's Complement

Note that this works both ways. If you have -30, and want to represent it in 2's complement, you take the binary representation of 30:

```
0000 0000 0000 0000 0000 0000 0001 1110
```

Invert the digits.

```
1111 1111 1111 1111 1111 1111 1110 0001
```

And add one.

```
1111 1111 1111 1111 1111 1111 1110 0010
```

Converted back into hex, this is 0xFFFFFFE2. And indeed, suppose you have this code:

```
#include <stdio.h>

int main() {
    int myInt;
    myInt = 0xFFFFFFE2;
    printf("%d\n",myInt);

    return 0;
}
```

That should yield an output of -30. Try it out if you like.

Arithmetic with Two's Complement

One of the nice properties of two's complement is that addition and subtraction is made very simple. With a system like two's complement, the circuitry for addition and subtraction can be unified, whereas otherwise they would have to be treated as separate operations.

In the examples in this section, I do addition and subtraction in two's complement, but you'll notice that every time I do actual operations with binary numbers I am always adding.

Example 1

Suppose we want to add two numbers 69 and 12 together. If we're to use decimal, we see the sum is 81. But let's use binary instead, since that's what the computer uses.

```
                                          1  1      Carry Row
  0000 0000 0000 0000 0000 0000 0100 0101 (69)
+ 0000 0000 0000 0000 0000 0000 0000 1100 (12)
-------------------------------------------------
  0000 0000 0000 0000 0000 0000 0101 0001 (81)
```

Example 2

Now suppose we want to subtract 12 from 69. Now, 69 - 12 = 69 + (-12). To get the negative of 12 we take its binary representation, invert, and add one.

```
0000 0000 0000 0000 0000 0000 0000 1100
```

Invert the digits.

```
1111 1111 1111 1111 1111 1111 1111 0011
```

And add one.

```
1111 1111 1111 1111 1111 1111 1111 0100
```

The last is the binary representation for -12. As before, we'll add the two numbers together.

```
   1111 1111 1111 1111 1111 1111 1    1    Carry Row
   0000 0000 0000 0000 0000 0000 0100 0101 (69)
+  1111 1111 1111 1111 1111 1111 1111 0100 (-12)
-------------------------------------------
   0000 0000 0000 0000 0000 0000 0011 1001 (57)
```

We result in 57, which is 69-12.

Example 3

Lastly, we'll subtract 69 from 12. Similar to our operation in example 2, 12 - 69 = 12 + (- 69). The two's complement representation of 69 is the following. I assume you've had enough illustrations of inverting and adding one.

```
1111 1111 1111 1111 1111 1111 1011 1011
```

So we add this number to 12.

```
                                    111       Carry Row
   0000 0000 0000 0000 0000 0000 0000 1100 (12)
+  1111 1111 1111 1111 1111 1111 1011 1011 (-69)
-------------------------------------------
   1111 1111 1111 1111 1111 1111 1100 0111 (-57)
```

This results in 12 - 69 = -57, which is correct.

Why Inversion and Adding One Works

Invert and add one. Invert and add one. It works, and you may want to know why. If you don't care, skip this, as it is hardly essential. This is only intended for those curious as to why that rather strange technique actually makes mathematical sense.

Inverting and adding one might sound like a stupid thing to do, but it's actually just a mathematical shortcut of a rather straightforward computation.

Borrowing and Subtraction

Remember the old trick we learned in first grade of "borrowing one's" from future ten's places to perform a subtraction? You may not, so I'll go over it. As an example, I'll do 93702 minus 58358.

```
  93702
- 58358
-------
```

Now, then, what's the answer to this computation? We'll start at the least significant digit, and subtract term by term. We can't subtract 8 from 2, so we'll borrow a digit from the next most significant place (the tens place) to make it 12 minus 8. 12 minus 8 is 4, and we note a 1 digit above the ten's column to signify that we must remember to subtract by one on the next iteration.

```
    1
  93702
- 58358
-------
    4
```

This next iteration is 0 minus 5, and minus 1, or 0 minus 6. Again, we can't do 0 minus 6, so we borrow from the next most significant figure once more to make that 10 minus 6, which is 4.

```
   11
  93702
- 58358
-------
   44
```

This next iteration is 7 minus 3, and minus 1, or 7 minus 4. This is 3. We don't have to borrow this time.

```
   11
  93702
- 58358
-------
  344
```

This next iteration is 3 minus 8. Again, we must borrow to make thi 13 minus 8, or 5.

```
  1 11
  93702
- 58358
-------
 5344
```

This next iteration is 9 minus 5, and minus 1, or 9 minus 6. This is 3. We don't have to borrow this time.

```
  1 11
  93702
- 58358
-------
  35344
```

So 93702 minus 58358 is 35344.

Borrowing and it's Relevance to the Negative of a Number

When you want to find the negative of a number, you take the number, and subtract it from zero. Now, suppose we're really stupid, like a computer, and instead of simply writing a negative sign in front of a number A when we subtract A from 0, we actually go through the steps of subtracting A from 0.

Take the following idiotic computation of 0 minus 3:

```
                 1           11          111          1111
  000000      000000      000000      000000      000000
  -    3      -    3      -    3      -    3      -    3
  ------      ------      ------      ------      ------
               7           97          997          9997
```

Et cetera, et cetera. We'd wind up with a number composed of a 7 in the one's digit, a 9 in every digit more significant than the 10^0's place.

The Same in Binary

We can do more or less the same thing with binary. In this example I use 8 bit binary numbers, but the principle is the same for both 8 bit binary numbers (chars) and 32 bit binary numbers (ints). I take the number 75 (in 8 bit binary that is 01001011_2) and subtract that from zero.

Sometimes I am in the position where I am subtracting 1 from zero, and also subtracting another borrowed 1 against it.

```
                      1              11              111             1111
  00000000          00000000        00000000        00000000        00000000
- 01001011        - 01001011      - 01001011      - 01001011      - 01001011
----------        ----------      ----------      ----------      ----------
                         1              01              101             0101

    11111           111111          1111111         11111111
  00000000          00000000        00000000        00000000
- 01001011        - 01001011      - 01001011      - 01001011
----------        ----------      ----------      ----------
     10101           110101          0110101         10110101
```

If we wanted we could go further, but there would be no point. Inside of a computer the result of this computation would be assigned to an eight bit variable, so any bits beyond the eighth would be discarded.

With the fact that we'll simply disregard any extra digits in mind, what difference would it make to the end result to have subtracted 01001011 from 100000000 (a one bit followed by 8 zero bits) rather than 0? There is none. If we do that, we wind up with the same result:

```
 11111111
 100000000
- 01001011
----------
 010110101
```

So to find the negative of an n-bit number in a computer, subtract the number from 0 or subtract it from 2^n. In binary, this power of two will be a one bit followed by n zero bits.

In the case of 8-bit numbers, it will answer just as well if we subtract our number from (1 + 11111111) rather than 100000000.

```
        1
+ 11111111
- 01001011
----------
```

In binary, when we subtract a number A from a number of all 1 bits, what we're doing is inverting the bits of A. So the subtract operation is the equivalent of inverting the bits of the number. Then, we add one.

So, to the computer, taking the negative of a number, that is, subtracting a number from 0, is the same as inverting the bits and adding one, which is where the trick comes from.
