Two's complement

description

It is curious how negative numbers are stored in most computers. The high value bit is reserved as a *sign* bit.

Although my machine and most others are 64-bit, we'll work with 1-byte (8-bit) examples for ease of representation.

With 8 bits, if the top bit is reserved to indicate sign (1 for negative and 0 for positive), there remain 7 bits to hold the actual value. Thus, on our 8-bit machine the largest and smallest positive integers are

```
0111 1111 = 127
0000 0001 =   1
```

So then for negative numbers, the high value bit is 1, as we've said. The rest is determined by our rule that any number plus its negative must equal zero.

```
0111 1111 =  127
1000 0001 = -127
---------
0000 0000
```

The *carry bit* propagates through in carrying out the addition and *overflows* off the end.

If you compare the digits of 127 and its negative, - 127, the bits are all

flipped, but then an extra 1 is added.

This also works with 1. To write -1, flip all the bits from 1, and then add 1 to the result.

```
0000 0001 =   1
1111 1111 =  -1
---------
0000 0000
```

It looks strange, but it works.

Adding two binary numbers with a carry row

An advantage of two's complement is that it unifies addition and subtraction so that we can use the same operations for both.

Let's compute 69 - 12. To do that we will add (-12) to 69. 69 = 64 + 4 + 1. So bin(69) is

```
0100 0101
```

And 12 = 8 + 4, so bin(12) is

```
0000 1100
```

We want minus 12. In two's complement, to obtain a negative number, invert the digits and add 1. bin(-12) is

```
1111 0011
0000 0001
---------
1111 0100
```

Adding 69 + (-12)

```
0100 0101
1111 0100
```

```
---------
0011 1001
```

Ignore the overflow. The result is 32 + 16 + 8 + 1 = 57

why this works

Suppose we compute -12 by subtracting 12 from 0.

```
0000 0000
0000 1100
---------
       00
```

At the third place we need to "borrow" a 1

```
     1
0000 0000
0000 1100
---------
      100
```

But that's not quite right. 0 doesn't have 1 to give. We must propagate the borrow. Introduce a *carry* row

```
1111 1
0000 0000
0000 1100
---------
      100
```

so to continue

```
1111 1
0000 0000
0000 1100
```

```
---------
1111 0100
```

That's what we had before.

look ahead

Another issue with the carry bit is that it's slow to do the operations one bit at a time.

We'd like to look at an addition problem and be able to say exactly where the carry bits will be. We use a 2-byte machine to make the example clearer.

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

Working from right to left, a carry bit is needed when both addends have 1, and not before.

Once a carry bit is present, it propagates when either one or both of the addends have 1.

Once we know the carry bits, just add all three values (by doing Carry OR A OR B), and disregard the carry in the result.