Fixed Point (Integers)

A **fixed point** representation is used to store integers. So we are interpreting a fixed number of bits to a number that we can understand. There are 4 ways for integer representation:

1. Unsigned Integer
2. Offset Binary
3. Sign and Magnitude
4. Two's Complement

Refer to the charts below as we explain the differences between these 4 representations.

UNSIGNED INTEGER		OFFSET BINARY		SIGN AND MAGNITUDE		TWO'S COMPLEMENT	
Decimal	Bit Pattern	Decimal	Bit Pattern	Decimal	Bit Pattern	Decimal	Bit Pattern
15	1111	8	1111	7	0111	7	0111
14	1110	7	1110	6	0110	6	0110
13	1101	6	1101	5	0101	5	0101
12	1100	5	1100	4	0100	4	0100
11	1011	4	1011	3	0011	3	0011
10	1010	3	1010	2	0010	2	0010
9	1001	2	1001	1	0001	1	0001
8	1000	1	1000	0	0000	0	0000
7	0111	0	0111	0	1000	-1	1111
6	0110	-1	0110	-1	1001	-2	1110
5	0101	-2	0101	-2	1010	-3	1101
4	0100	-3	0100	-3	1011	-4	1100
3	0011	-4	0011	-4	1100	-5	1011
2	0010	-5	0010	-5	1101	-6	1010
1	0001	-6	0001	-6	1110	-7	1001
0	0000	-7	0000	-7	1111	-8	1000

Unsigned Integer

The **unsigned integer** is a simple binary format. Negative numbers cannot be represented in this format.

Adding and Subtracting

Binary math on paper can be accomplished in multiple ways. One way is to convert the binary numbers to decimal, complete the computation, and return the result back to binary. Below are examples of how to perform these computations staying only in binary.

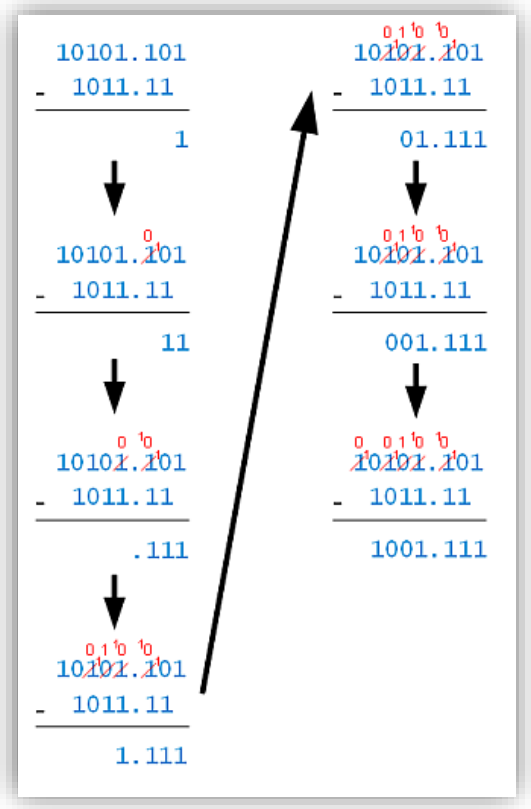
Adding

Just like decimal numbers, when adding binary numbers, add the numbers straight down, carrying over a 1 when necessary, very similar to carrying the 1 in decimal addition.

		1	1 1
000	010	001	011
+101	+101	+001	+011
101	111	010	110

Subtracting

For subtracting, sometimes lesser values need to borrow from greater values, similar to subtraction in decimal. To the right is an example and included steps for binary subtraction.

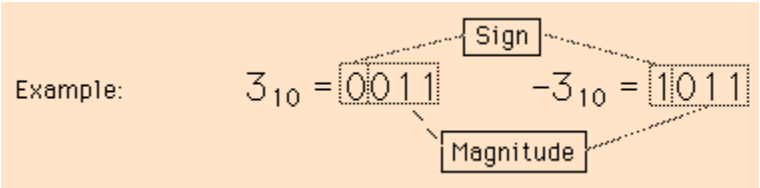


1. 1 – 0 = 0
2. Borrow to make 10 – 1 = 1
3. Borrow to make 10 – 1 = 1
4. Cascaded borrow to make 10 – 1 = 1
5. 1 – 1 = 0
6. 0 – 0 = 0
7. Borrow to make 10 – 1 = 1

Sign and Magnitude

The **sign and magnitude** allows for negative numbers. The first bit on the left side signifies whether the representation is negative or positive. A 0 will represent a positive number or 0 and a 1 will represent a negative number or 0. The range of numbers that can be achieved would be half of the largest number (e.g. 4-bit largest number is 15, so the range will be from -7 to +7). This is the only number system with two zeros as well (+0 and -0 are just 0).

The most significant bit (left-most bit) of a binary number can be used to represent the **sign** of the number. The other bits are used to represent the **magnitude**. Below is an example of showing the values "3" and "-3". Notice that the magnitude is the same for both binary numbers, but the sign reflects whether it is negative or positive.



Offset Binary

Offset binary is commonly used in image processing to display negative pixels' values. The offset binary allows for negative numbers. The first bit on the left side signifies whether the representation is negative or positive. A 1 will represent a positive number and a 0 will represent a negative number or 0.

This is useful for binary counters. The magnitude revolves around a "**Working zero**". For example:

1. Start with 4 bits, and the highest you can go with four bits is 15 (1111).
2. Take half of your highest number, in this case it will be 7. (round down)
3. Now to find out the Working zero for 4 bits, we must subtract half from the largest number.

Largest number (15)	1111
Subtract half (7)	-0111
Working zero	1000

So we know that 0 for a 4-bit offset binary will be "1000"

4. Now if we want “+4” or “-4”, we just have to add or subtract from our working 0.

To get +4	1000	To get -4	1000
	+0100		-0100
	1100		0011

Here is another example, but I changed it to 8 bits.

1. Maximum number is 11111111 which is 255 (decimal).
2. Half of that is 127 (round down).
3. Working zero:

Largest number (255)	1111 1111
Subtract half (127)	-0111 1111
Working zero	**1000 0000**

4. Now, if I want “+101” and “-23”

To get +101	1000 0000 [working zero]	To get -23	1000 0000
	+0110 0101 [101 in binary]		-0001 1001
	1110 0101 [101 in offset binary]		0111 1000

Two’s Complement

Two’s complement is the easiest for hardware design and the most common format for general purpose computing. Two’s complement allows for negative numbers. The first bit on the left side signifies whether the representation is negative or positive. A 0 will represent a positive number or 0 and a 1 will represent a negative number. This format has the same range as offset binary and sign and magnitude but it is more common to see this format in a question or lesson plan than the other two formats.

To produce Two’s complement numbers are in two parts, positive and negative.

Positive numbers (including 0) are similar to unsigned integer (binary), they are written the same way.

Negative numbers have a few extra steps. For example, to produce the number -4:

1. Write the binary number for 4	0100
2. Take the complement of it	1011
3. Add 1	1100

Another example for -7:

1. Write the binary number for 7	0111
2. Take the complement of it	1000
3. Add 1	1001

Overflow

Overflow is an error that occurs when the resulting value of an operation performed on valid representations of numbers is out of the range of valid values. That is, the resulting value is greater than max or less than min. So you may have heard of a 32-bit system or a 64-bit system, this is the number of bits used in computation and when a computation goes above the fixed bits, an error will occur. Let’s do an example but with an 8-bit system.

So let's add 150 and 249:

```
  150
+249
  399
```

The answer is 399, but we need to do this in binary.

```
                1111
[150]        1001 0110
[249]       +1111 1001
            11000 1111
```

Since this is an 8-bit system, an error is thrown due to the answer being 9-bits. In some systems, the extra bit (highlighted in green) is simply ignored and returns the rest. In this case, 1000 1111 is 143.

As we already know, 150 + 249 = 399, not 143. This result is an overflow error.