CS1800
Fall 2026
Two's Complement

In our first tour through number representation, we assumed all values were unsigned. Let's wrap in negative numbers now and let's turn our attention back to the binary number system.

In a real computer, we have physical limitations in terms of how much data we can store about a number. Typically, an integer is allocated a certain number of bits, often 32 or 64.

Sign and Magnitude
The simplest way to represent negative numbers would be to designate one bit as the sign bit and leave it as that:

1	1	0	1	1	0	1	1
Sign bit	Magnitude						

This number would be negative 91_{10} (the sign bit uses 1 for negative numbers and 0 for positive).

This system, "sign and magnitude", is OK... until we have to start adding and subtracting numbers. We saw in class how to add two binary numbers together; computers are pretty good at that, and we use for subtraction as well as addition (computers will see $x - y$ as $x + \ - y$). This is where sign and magnitude screws us up. Let's say we want to do $2 - 3$:

```
  0 0 0 0 0 0 1 0      2₁₀ in sign-and-magnitude format
+ 1 0 0 0 0 0 1 1      -3₁₀ in sign-and-magnitude format
 -----------------
  1 0 0 0 0 1 0 1      -5₁₀ in sign-and-magnitude format :(
```

By applying just normal addition, *2 + -3*, we **should have** gotten the result *-1* but instead we got *-5* (why???). So annoying. Plain old sign-and-magnitude is broken. Two's complement is meant to fix that.

Two's Complement

In **two's complement**, the positive values are the same as in sign-and-magnitude. It's the negative values that are different. To convert a negative decimal number to two's complement, take the following steps:

1. Represent the magnitude of the number in binary, unsigned
2. Flip the bits
3. Add one

Let's say we're in 8-bit two's complement, and try a few conversions:

Example: Decimal number: -1_{10}

1. Represent as unsigned in 8-bit binary: `0 0 0 0 0 0 0 1`
2. Flip the bits: `1 1 1 1 1 1 1 0`
3. Add one: `1 1 1 1 1 1 1 1`

Example: Decimal number: -2_{10}

1. Represent as unsigned in 8-bit binary: `0 0 0 0 0 0 1 0`
2. Flip the bits: `1 1 1 1 1 1 0 1`
3. Add one: `1 1 1 1 1 1 1 0`

Example: Decimal number: -5_{10}

4. Represent as unsigned in 8-bit binary: `0 0 0 0 0 1 0 1`
5. Flip the bits: `1 1 1 1 1 0 1 0`
6. Add one: `1 1 1 1 1 0 1 1`

Example: Decimal number: -127_{10}

7. Represent as unsigned in 8-bit binary: `0 1 1 1 1 1 1 1`
8. Flip the bits: `1 0 0 0 0 0 0 0`
9. Add one: `1 0 0 0 0 0 0 1`

Does this fix the problem we saw with sign-and-magnitude? Let's find out! Let's say we want to do $2 - 3$, but with two's complement this time.

```
  0 0 0 0 0 0 1 0      2₁₀ in two's complement
+ 1 1 1 1 1 1 0 1      -3₁₀ in two's complement
 -----------------
  1 1 1 1 1 1 1 1      -1₁₀ in two's complement :)
```

That give us -1 as expected -- excellent! Problem solved!

If we list out all the positive/negative two's complement numbers in order, you can see how the pattern starts to emerge:

```
0 1 1 1 1 1 1 1   127
0 1 1 1 1 1 1 0
0 1 1 1 1 1 0 1   125
...
0 0 0 0 0 0 1 0     2
0 0 0 0 0 0 0 1     1
0 0 0 0 0 0 0 0     0
1 1 1 1 1 1 1 1    -1
1 1 1 1 1 1 1 0    -2
1 1 1 1 1 1 0 1    -3
1 1 1 1 1 1 0 0    -4
```

```
1 1 1 1 1 0 1 1    -5
...
1 0 0 0 0 0 1 1  -125
1 0 0 0 0 0 1 0  -126
1 0 0 0 0 0 0 1  -127
1 0 0 0 0 0 0 0  -128
```

Overflow in Two's Complement

In two's complement, *overflow* occurs when we try to store a number that is too high (positive) or too low (negative) to fit in the specified number of bits. Two's complement reflects the real-world of computing, and in the real world we don't have infinite bits to store our values. Once we run out of bits, we're just out!

Overflow is not the same as just having an extra bit after an arithmetic operation. In that case we always chop off the extra bit. No matter what -- *always chop off the extra bit!* You might end up in overflow, or not, but always lose the extra bit. Once we've completed the arithmetic operation, including dropping the extra bit if necessary, *then* we look for overflow.

Detecting overflow
There are two ways to check for overflow in an arithmetic operation:

1. **Convert to decimal and check.** Determine the highest positive and lowest negative values you can store in the number of bits (we just saw in 8-bit two's complement that we can store from -128 to 127). If we're adding two values in 8-bit two's complement and the result would be greater than 127 or lower than -128, that is overflow!
2. **Proceed in two's complement and check the result.** Perform the arithmetic operation on the two's complement values. If you end up with an extra bit, chop it off (always!). Then, check the sign of the resulting value. If you added two positive values and ended up with a negative, or vice-versa, that's a sign of overflow.

Overflow / Not Overflow examples (4-bit two's complement)

Example 1

```
   0 1 0 1
+  0 0 1 0
  ---------
   0 1 1 1
```

This is NOT overflow! We know because...

- Converted to decimal, we're adding 5+ 2 = 7, and 7 fits in 4-bit two's complement.
- AND, we added two positive values here and ended up with a positive.

Example 2

```
   0 1 1 1
+  0 0 1 0
___________
   1 0 0 1
```

This is overflow! We know because...

- Converted to decimal, we're adding 7 + 2 = 9, and 9 is too big to store in 4-bit two's complement.
- AND, we added two positive values here and ended up with a negative.

Example 3

```
   1 1 0 1
+  1 1 1 0
___________
 1 1 0 1 1
Chop off the extra bit: 1 0 1 1
```

This is NOT overflow! We know because...

- Converted to decimal, we're adding -3 + -2 = -5, and -5 fits in 4-bit two's complement.
- We chop off the extra bit and get a negative value (`1 0 1 1`), which makes sense because we've added two negatives and got a negative.

Example 4

```
   1 0 1 1
+  1 1 0 0
___________
 1 0 1 1 1
Chop off the extra bit:  0 1 1 1
```

This is overflow! We know because...

- Converted to decimal, we're adding -5 + -4 = -9, and -9 does not fit in 4-bit two's complement.
- AND, we chop off the extra bit and get a positive value (`0 1 1 1`), even though we've added two negatives.