# Section 2: Integer & Floating Point Numbers

- <u>**Representation of integers: unsigned and signed**</u>
- **Unsigned and signed integers in C**
- **Arithmetic and shifting**
- **Sign extension**

- **Background: fractional binary numbers**
- **IEEE floating-point standard**
- **Floating-point operations and rounding**
- **Floating-point in C**

# Unsigned Integers

- **Unsigned values are just what you expect**
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_7 2^7 + b_6 2^6 + b_5 2^5 + \ldots + b_1 2^1 + b_0 2^0$
    - Useful formula: $1+2+4+8+\ldots+2^{N-1} = 2^N - 1$

- **You add/subtract them using the normal "carry/borrow" rules, just in binary**

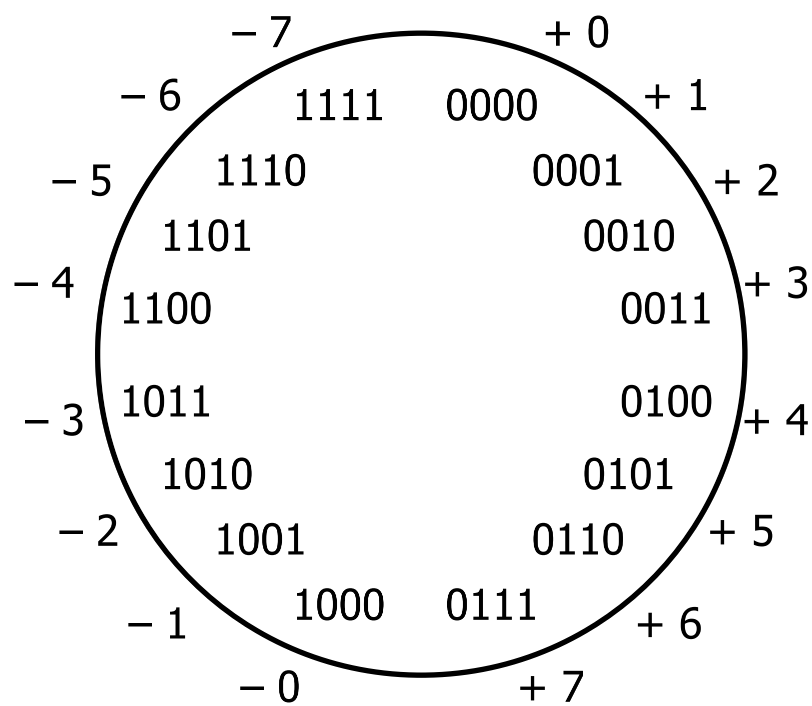
```
  63       00111111
+  8      +00001000
  71       01000111
```

# Signed Integers

- **Let's do the natural thing for the positives**
  - They correspond to the unsigned integers of the same value
    - Example (8 bits): 0x00 = 0, 0x01 = 1, …, 0x7F = 127
- **But, we need to let about half of them be negative**
  - Use the high order bit to indicate *negative*: call it the "sign bit"
    - Call this a "sign-and-magnitude" representation
  - Examples (8 bits):
    - 0x00 = $00000000_2$ is non-negative, because the sign bit is 0
    - 0x7F = $01111111_2$ is non-negative
    - 0x85 = $10000101_2$ is negative
    - 0x80 = $10000000_2$ is negative…

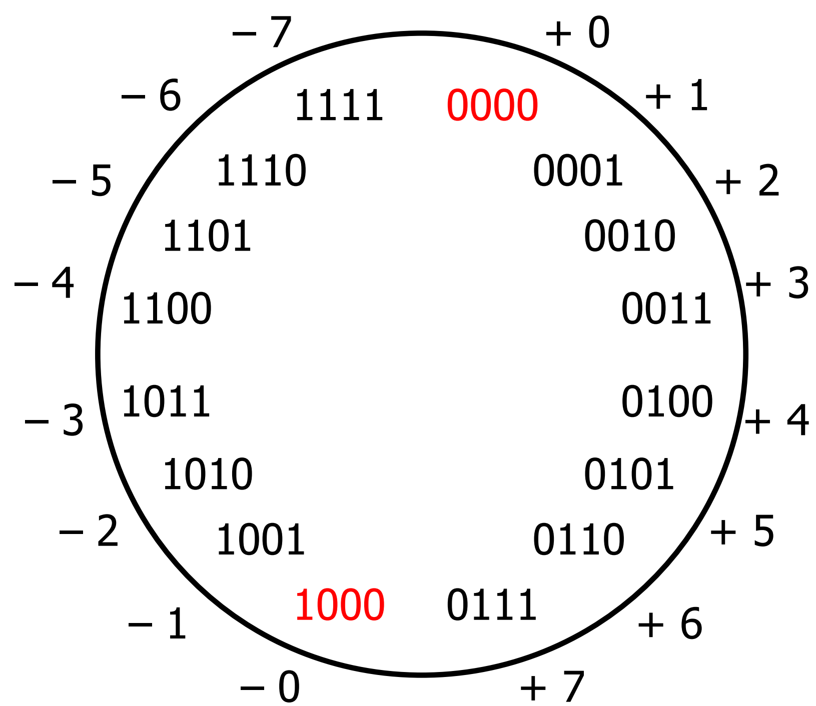
# Sign-and-Magnitude Negatives

- **How should we represent -1 in binary?**
  - Sign-and-magnitude: $10000001_2$
    Use the MSB for + or -, and the other bits to give magnitude



# Sign-and-Magnitude Negatives

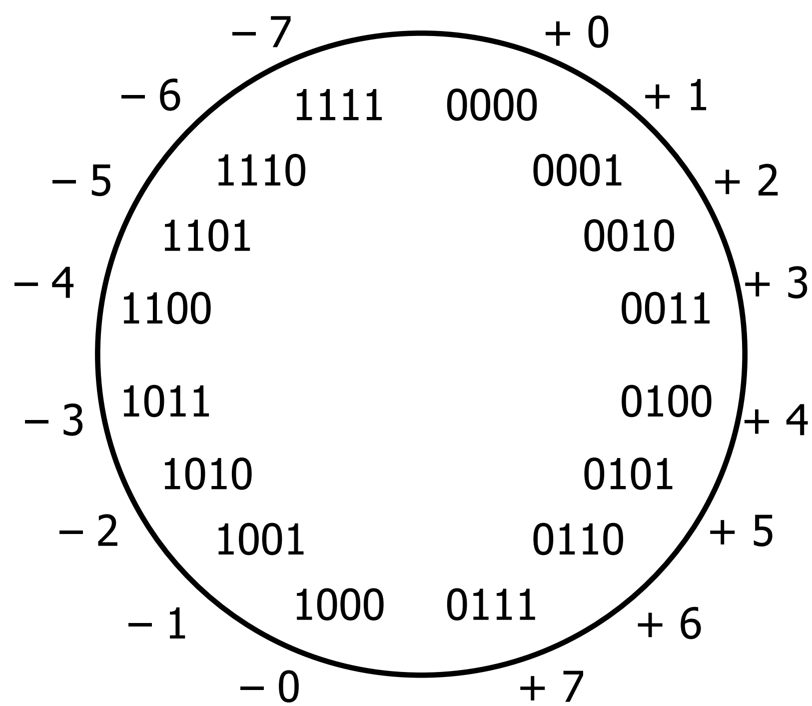
- **How should we represent -1 in binary?**
  - Sign-and-magnitude: $10000001_2$
    Use the MSB for + or -, and the other bits to give magnitude
    (Unfortunate side effect: there are two representations of 0!)



# Sign-and-Magnitude Negatives

- **How should we represent -1 in binary?**
  - Sign-and-magnitude: $10000001_2$
    Use the MSB for + or -, and the other bits to give magnitude
    (Unfortunate side effect: there are two representations of 0!)
  - Another problem: math is cumbersome
    - Example:
      4 - 3 != 4 + (-3)

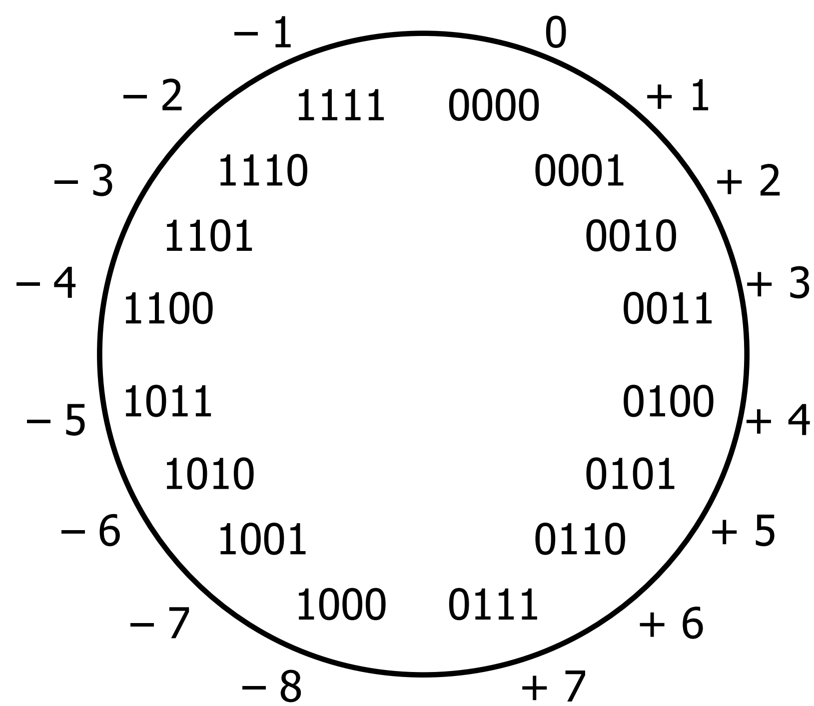
```
  0100
+ 1011
  ----
  1111
```



# Two's Complement Negatives

- **How should we represent -1 in binary?**
  - Rather than a sign bit, let MSB have same value, but *negative* weight
    - W-bit word: Bits 0, 1, …, W-2 add $2^0, 2^1, \ldots, 2^{W-2}$ to value of integer when set, but bit W-1 adds $-2^{W-1}$ when set
    - e.g. unsigned $1010_2$: $1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 10_{10}$
      2's comp. $1010_2$: $-1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = -6_{10}$
  - So -1 represented as $1111_2$; all negative integers still have MSB = 1
  - Advantages of two's complement: only one zero, simple arithmetic
  - To get negative representation of any integer, take bitwise complement and then add one!

```
~x + 1 = -x
```



# Two's Complement Arithmetic

- **The same addition procedure works for both unsigned and two's complement integers**
  - Simplifies hardware: only one adder needed
  - Algorithm: simple addition, discard the highest carry bit
    - Called "modular" addition: result is sum *modulo* $2^W$
- **Examples:**

| | | | | | |
|---|---|---|---|---|---|
| 4 | 0100 | 4 | 0100 | – 4 | 1100 |
| + 3 | + 0011 | – 3 | + 1101 | + 3 | + 0011 |
| = 7 | = 0111 | = 1 | 1 0001 | – 1 | 1111 |
| | | drop carry | = 0001 | | |

# Two's Complement

- **Why does it work?**
  - Put another way: given the bit representation of a positive integer, we want the negative bit representation to always sum to 0 (ignoring the carry-out bit) when added to the positive representation
  - This turns out to be the *bitwise complement plus one*
    - What should the 8-bit representation of -1 be?

```
 00000001
+????????     (we want whichever bit string gives the right result)
---------
 00000000

 00000010       00000011
+????????      +????????
---------      ---------
 00000000       00000000
```

# Two's Complement

- **Why does it work?**
  - Put another way: given the bit representation of a positive integer, we want the negative bit representation to always sum to 0 (ignoring the carry-out bit) when added to the positive representation
  - This turns out to be the *bitwise complement plus one*
    - What should the 8-bit representation of -1 be?

```
 00000001
+11111111    (we want whichever bit string gives the right result)
---------
100000000

 00000010      00000011
+????????     +????????
---------     ---------
 00000000      00000000
```

# Two's Complement

- **Why does it work?**
  - Put another way: given the bit representation of a positive integer, we want the negative bit representation to always sum to 0 (ignoring the carry-out bit) when added to the positive representation
  - This turns out to be the *bitwise complement plus one*
    - What should the 8-bit representation of -1 be?

```
 00000001
+11111111     (we want whichever bit string gives the right result)
---------
100000000

 00000010      00000011
+11111110     +11111101
---------     ---------
100000000     100000000
```

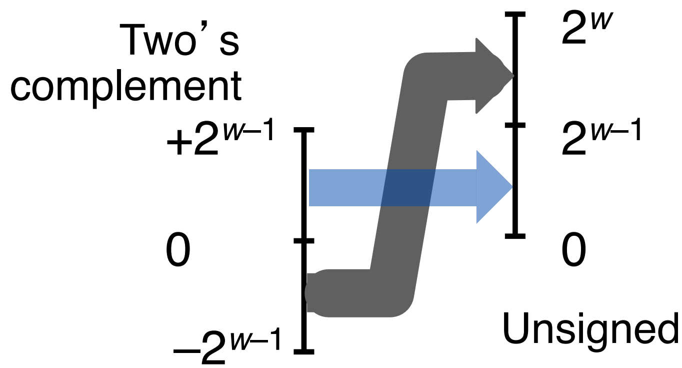
# Unsigned & Signed Numeric Values

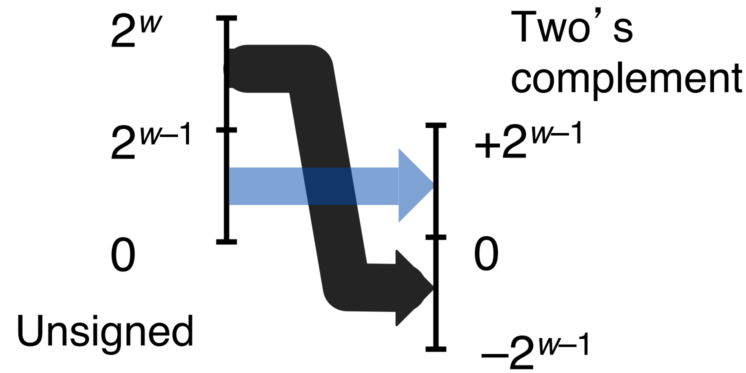
| *X* | Unsigned | Signed |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |
| 1000 | 8 | –8 |
| 1001 | 9 | –7 |
| 1010 | 10 | –6 |
| 1011 | 11 | –5 |
| 1100 | 12 | –4 |
| 1101 | 13 | –3 |
| 1110 | 14 | –2 |
| 1111 | 15 | –1 |

- Both signed and unsigned integers have limits
  - If you compute a number that is too big, you wrap: 6 + 4 = ? 15U + 2U = ?
  - If you compute a number that is too small, you wrap: -7 - 3 = ? 0U - 2U = ?
- The CPU may be capable of "throwing an exception" for overflow on signed values
  - But it won't for unsigned
- C and Java just cruise along silently when overflow occurs...

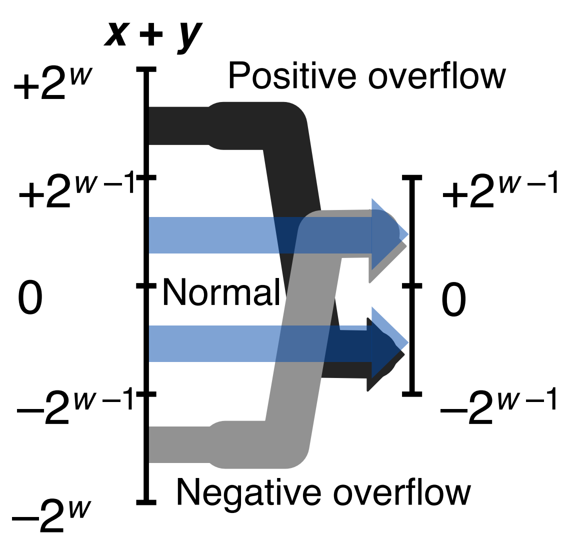
# Visualizations

- **Same W bits interpreted as signed vs. unsigned:**




- **Two's complement (signed) addition: x and y are W bits wide**



# Values To Remember

- **Unsigned Values**
  - UMin = 0
    - 000…0
  - UMax = $2^w - 1$
    - 111…1
- **Two's Complement Values**
  - TMin = $-2^{w-1}$
    - 100…0
  - TMax = $2^{w-1} - 1$
    - 011…1
  - Negative 1
    - 111…1 0xFFFFFFFF (32 bits)

**Values for $W = 16$**

| | Decimal | Hex | Binary |
|---|---:|---|---|
| `UMax` | 65535 | `FF FF` | `11111111 11111111` |
| `TMax` | 32767 | `7F FF` | `01111111 11111111` |
| `TMin` | -32768 | `80 00` | `10000000 00000000` |
| `-1` | -1 | `FF FF` | `11111111 11111111` |
| `0` | 0 | `00 00` | `00000000 00000000` |