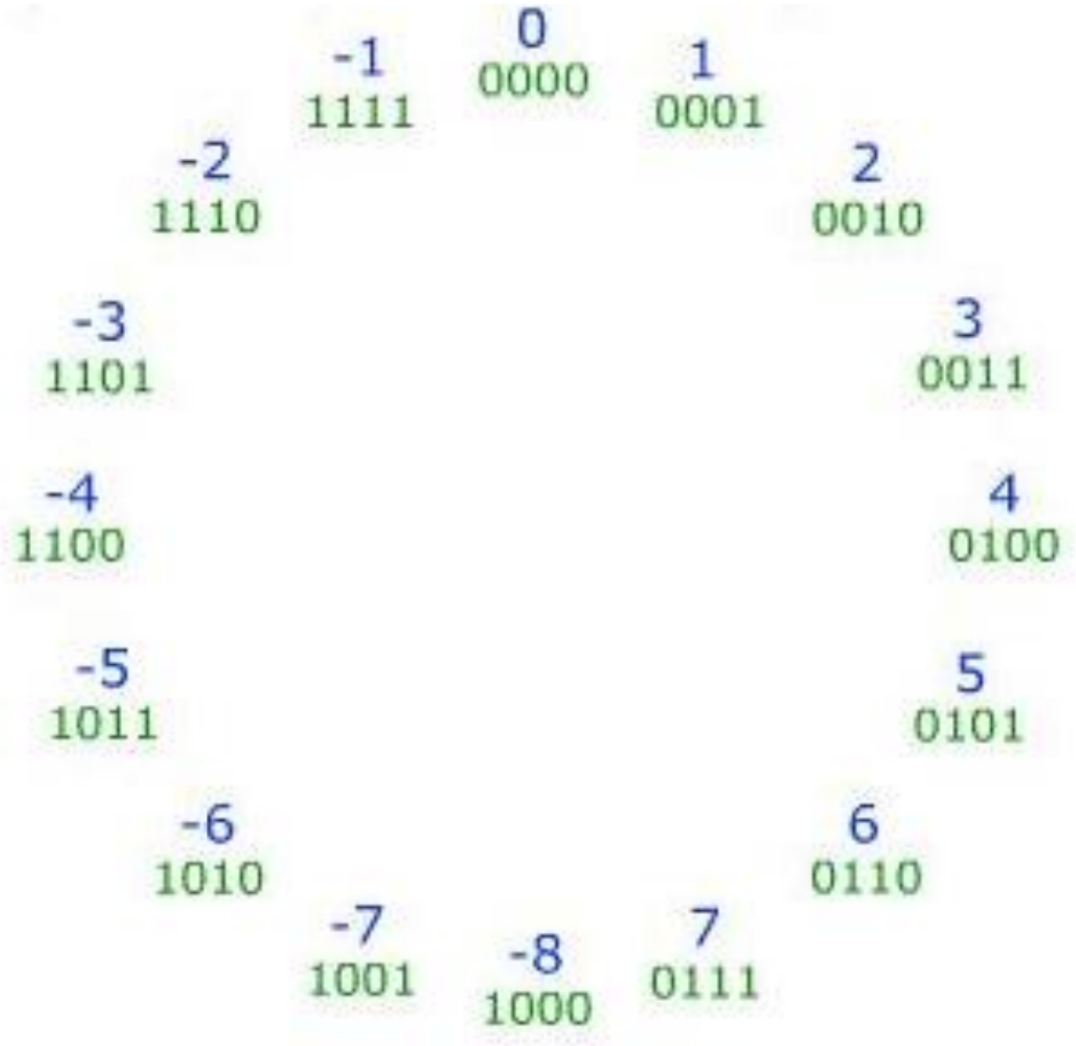
2’s Complement and Floating-Point

CSE 351 Section 3

Two's Complement

- An n-bit, two's complement number can represent the range $[-2^{n-1}, 2^{n-1} - 1]$.
  - Note the asymmetry of this range about 0 – there's one more negative number than positive
- Note what happens when you overflow



4-bit two's complement range

Understanding Two's Complement

- An easier way to find the decimal value of a two's complement number:

$$\sim x + 1 = -x$$

- We can rewrite this as x = ~(-x - 1), i.e. subtract 1 from the given number, and flip the bits to get the positive portion of the number.

- Example: `0b11010110`
  - Subtract 1: `0b11010110 - 1 = 0b11010101`
  - Flip the bits: `~0b11010101 = 0b00101010`
  - Convert to decimal: $0b00101010 = (32+8+2)_{10} = 42_{10}$
  - Multiply by negative one, Answer: -42_{10}.

Two's Complement: Operations

```
int x = 0xAB;
int y = 17;
int z = 5;
int result = ~(x | y) + z;
```

What is the value of result in decimal?

1. Convert numbers to binary
   - `0xAB = 0b10101011`
   - 17_{10} = `0b00010001`
2. Compute `x | y`

```
  0000 ... 1010 1011
| 0000 ... 0001 0001
--------------------
  0000 ... 1011 1011
```

3. Compute `~(x | y)` (flip the bits)

```
~ 0000 0000 0000 0000 0000 0000 1011 1011
-----------------------------------------
  1111 1111 1111 1111 1111 1111 0100 0100
```

Two's Complement: Operations

```
int x = 0xAB;
int y = 17;
int z = 5;
int result = ~(x | y) + z;
```

What is the value of result in decimal?

4. Compute `~(x | y) + z`

   - You can either convert to decimal before or after adding z
   - 17_{10} = 0b00010001
   - Convert $11111111111111111111111101000100_2$ to decimal (hint: -x = ~x + 1)
   - $\sim 11111111111111111111111101000100_2 = 00000000000000000000000010111011_2$
   - $\sim 0000\ldots\ldots 10111011_2 + 1 = 0000\ldots\ldots 10111100_2$
   - $-x = 0000\ldots\ldots 10111100_2$
   - $x = 1000\ldots\ldots 10111100_2$

Floating Point

- value = $(-1)^{s} * M * 2^{E}$
- Numerical Form
  - Sign bit **s** determines whether number is negative or positive
  - Significand (mantissa) **M** normally a fractional value in range [1.0, 2.0)
  - Exponent **E** weights value by a (possibly negative) power of two
- Representation in Memory
  - MSB s is sign bit **s**
  - exp field encodes **E** (but is ***not equal*** to E) – remember the bias
  - Frac field encodes **M** (but is ***not equal*** to M)

Floating Point

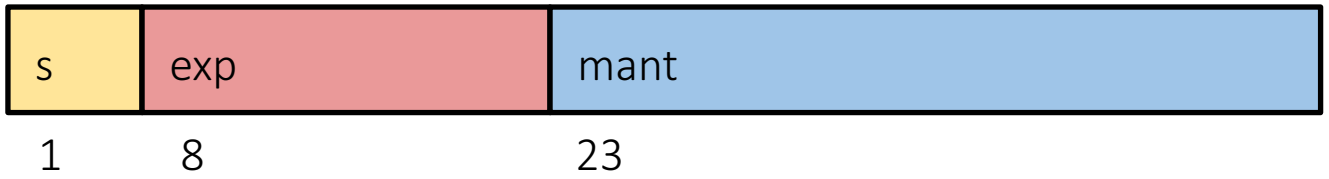
- value = $(-1)^{S} * M * 2^{E}$
- Value: ±1 × Mantissa × 2^{Exponent}
- Bit Fields: $(-1)^{S} \times 1.M \times 2^{(E+\text{bias})}$
- Bias
  - Read exponent as unsigned, but with bias of $-(2^{w-1}-1) = -127$
  - Representable exponents roughly ½ positive and ½ negative
  - Exponent 0 (Exp = 0) is represented as E = 0b 0111 1111
- Why?
  - Floating point arithmetic = easier
  - Somewhat compatible with 2's complement

Floating Point

- Exponent overflow yields +∞ or -∞
- Floats with value +∞, -∞, and NaN can be used in operations
- Result usually still +∞, -∞, or NaN; sometimes intuitive, sometimes not

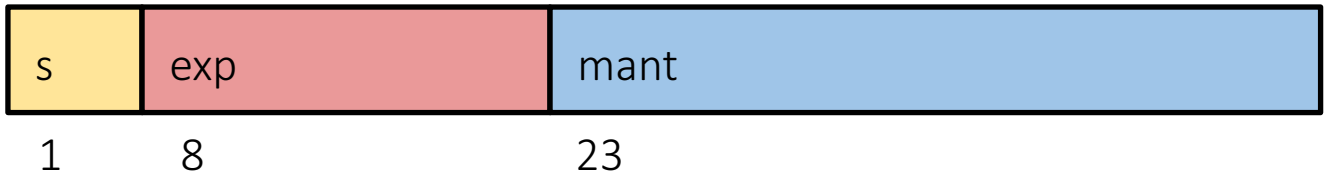
- Floating point ops do not work like real math, due to rounding!
- Not associative:
  - (3.14 + 1e100) – 1e100 != 3.14 + (1e100 – 1e100)
- Not distributive:
  - 100 * (0.1 + 0.2) != 100 * 0.1 + 100 * 0.2
- Not cumulative
  - Repeatedly adding a very small number to a large one may do nothing

Floating Point Examples



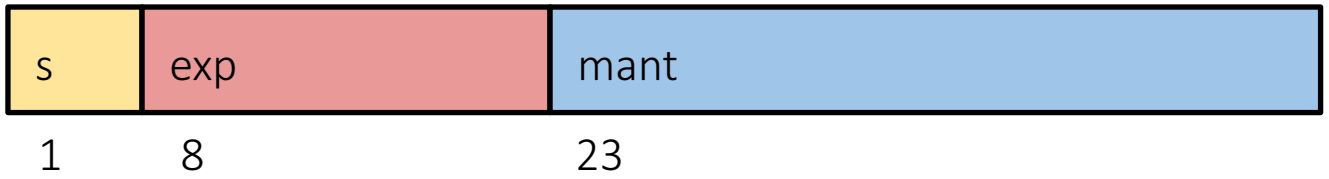
- How do you represent -1.5 in floating point?
  - Sign bit: 1
  - First the integral part of the value: 1 = 0b1
  - Now compute the decimal: 0.5 = 0b0.1
  - 1.5_{10} = 1.1b
  - Don't need to normalize because it's already in scientific notation: 1.1×2^0
  - Exponent: $0 + 127 = 127_{10} = 01111111_2$
  - Mantissa: $10000000000000000000000_2$
  - -1.510 = 1 01111111 $10000000000000000000000_2$ = 0xBFC00000

Floating Point Examples



- How do you represent 1.0 in floating point?
  - Sign bit: 0
  - First the integral part of the value: 1 = 0b1
  - 1.0_{10} => 1.0b
  - Don't need to normalize because it's already in scientific notation: 1.0×2^0
  - Exponent: $0 + 127 = 127_{10} = 01111111_2$
  - Mantissa: $00000000000000000000000_2$
  - $1.010 = 0\ 01111111\ 00000000000000000000000_2 = 0x3F800000$

Floating Point Addition



$(-1)^{s1} * M1 * 2^{E1} \; + \; (-1)^{s2} * M2 * 2^{E2}$

(Assume *E1* > *E2)*

- Exact Result: $(-1)^{s} * M * 2^{E}$
  - Sign `s`, mantissa `M`:
    - M = M1 + M2, result of signed align & add
  - Exponent `E`: `E1`

- Fixing
  - If *M* ≥ 2, shift *M* right, increment *E*
  - if *M* < 1, shift *M* left *k* positions, decrement *E* by *k*
  - Overflow if *E* out of range
  - Round *M* to fit `frac` precision

What is floating point result of 12.3125 + 1.5?

12.3125_{10} = `0 10000010 10001010000000000000000`

1.5_{10} = `0 01111111 10000000000000000000000`

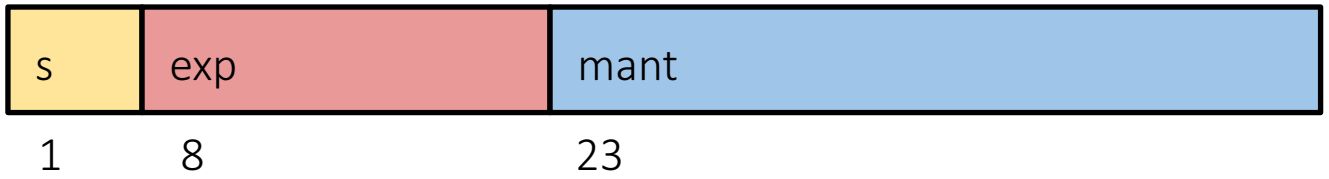
In scientific notation: $(1.1000101 \times 2^3) + (1.1 \times 2^0)$

```
   1100.0101 x 2^0
+     1.1    x 2^0
-----------------
   1101.1101 x 2^0
```

Adjusted mantissa: 1.1011101 => M = 1011101

Ans: `0 10000010 10111010000000000000000`

Floating Point Multiplication



$(-1)^{s1} * M1 * 2^{E1} \quad * \quad (-1)^{s2} * M2 * 2^{E2}$

- Exact Result: $(-1)^{s} * M * 2^{E}$
  - Sign `s`: `s1 ^ s2`
  - Mantissa `M`: `M1 * M2`
  - Exponent `E`: `E1 + E2`

- Fixing
  - If *M* ≥ 2, shift *M* right, increment *E*
  - If *E* out of range, overflow
  - Round *M* to fit `frac` precision

What is floating point result of 1.1* 1.0?

1.1_{10} = `0 01111111 00011001100110011001101`

1.0_{10} = `0 01111111 00000000000000000000000`

In scientific notation: $(1.1 \times 2^0) + (1.0 \times 2^0)$

```
      1.00011001100110011001101 x 2⁰
*                          1.0  x 2⁰
-----------------------------------
      00000000000000000000000000
      10001100110011001100110100
Result: 1.00011001100110011001101 x 2⁰
```

Ans: `0 01111111 00011001100110011001101`

Floating Point Worksheet