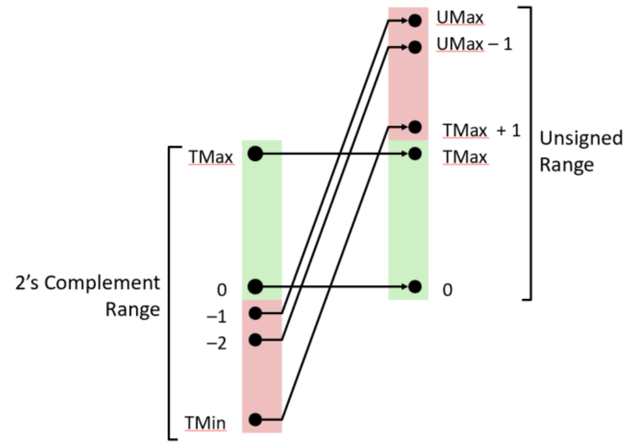
CSE 351 Section 3 – Integers and Floating Point

Welcome back to section, we're happy that you're here ☺ .. . .

Integers and Arithmetic Overflow

Arithmetic overflow occurs when the result of a calculation can't be represented in the current encoding scheme (*i.e.*, it lies outside of the representable range of values), resulting in an incorrect value.

- **Unsigned overflow:** the result lies outside of [UMin, UMax]; an indicator of this is when you add two numbers and the result is smaller than either number.
- **Signed overflow:** the result lies outside of [TMin, TMax]; an indicator of this is when you add two numbers with the same sign and the result has the opposite sign.



<u>Exercises</u>:

1) Assuming these are all signed two's complement 6-bit integers, compute the result of each of the following additions. For each, indicate if it resulted in overflow. [Spring 2016 Midterm 1C]

001001	110001	011001	101111
+ 110110	+ 111011	+ 001100	+ 011111
111111	~~1~~ 101100	100101	~~1~~ 001110
No	No	Yes	No

2) Find the largest 8-bit unsigned numeral (answer in hex) such that c + 0x80 causes NEITHER signed nor unsigned overflow in 8 bits. [Autumn 2019 Midterm 1C]

   Unsigned overflow will occur for c > 0x80. Signed overflow can only happen if c is negative (also > 0x80). Largest is therefore, 0x7F

3) Find the smallest 8-bit numeral (answer in hex) such that c + 0x71 causes signed overflow, but NOT unsigned overflow in 8 bits. [Autumn 2018 Midterm 1C]

   For signed overflow, need (+) + (+) = (−). For no unsigned overflow, need no carryout from MSB. The first (−) encoding we can reach from 0x71 is 0x80. 0x80 – 0x71 = 0xF.

Goals of Floating Point

Representation should include: [1] a large range of values (both very small and very large numbers), [2] a high amount of precision, and [3] real arithmetic results (*e.g.* ∞ and NaN).

IEEE 754 Floating Point Standard

The value of a real number can be represented in scientific binary notation as:

$$\text{Value} = (-1)^{\text{sign}} \times \text{Mantissa}_2 \times 2^{\text{Exponent}} = (-1)^S \times 1.M_2 \times 2^{E\text{-bias}}$$

The binary representation for floating point values uses three fields:

- **S**: encodes the *sign* of the number (0 for positive, 1 for negative)
- **E**: encodes the *exponent* in **biased notation** with a bias of 2^{w-1}-1
- **M**: encodes the *mantissa* (or *significand*, or *fraction*) – stores the fractional portion, but does not include the implicit leading 1.

	S	E	M
`float`	1 bit	8 bits	23 bits
`double`	1 bit	11 bits	52 bits

How a `float` is interpreted depends on the values in the exponent and mantissa fields:

E	M	Meaning
0	anything	denormalized number (denorm)
1-254	anything	normalized number
255	zero	infinity (∞)
255	nonzero	not-a-number (NaN)

Exercises:

Bias Notation

1) Suppose that instead of 8 bits, E was only designated 4 bits. What is the bias in this case? $2^{(4 - 1)} - 1 = 7$

2) Compare these two representations of E for the following values:

Exponent	E (4 bits)	E (8 bits)
1	1 0 0 0	1 0 0 0 0 0 0 0
0	0 1 1 1	0 1 1 1 1 1 1 1
-1	0 1 1 0	0 1 1 1 1 1 1 0

Notice any patterns?

The representations are the same except the length of number of repeating bits in the middle are different.

Floating Point / Decimal Conversions

3) Let's say that we want to represent the number 3145728.125 (broken down as $2^{21} + 2^{20} + 2^{-3}$)

a. Convert this number to into single precision floating point representation:

0	1	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

b. How does this number highlight a limitation of floating point representation?
Could only represent 2^21 + 2^20. Not enough bits in the mantissa to hold 2^-3, which caused *rounding*.

4) What are the decimal values of the following `float`s?

0x80000000	0xFF94BEEF	0x41180000
–0	NaN	+9.5

0x41180000 = 0b 0|100 0001 0|001 1000 0…0.
S = 0, E = 128+2 = 130 → Exponent = E – bias = 3, Mantissa = 1.0011_2
$1.0011_2 \times 2^3 = 1001.1_2 = 8 + 1 + 0.5 = 9.5$

5) [Summer 2018 Midterm 1E-G] For the rest of this problem we are working with a new floating point datatype (**`flo`**) that follows the same conventions as IEEE 754 except using 8 bits split into the following fields:

Sign (1)	Exponent (3)	Mantissa (4)

E) What is the value of the numeral **`0b 1010 1000`** in this representation?

S = 1, E = 0b010, M = 0b1000. Bias = $2^{3-1} - 1 = 3$
$(-1)^1 \times 1.1000_2 \times 2^{2-3} = -1.1_2 \times 2^{-1} = -0.11_2 = -(0.5 + 0.25) = -0.75$

F) What is the *encoding* of the **most negative real number that we can represent** (∞ is not a real number) in this floating point scheme (binary)?

Largest normalized number, but negative: 0b11101111

G) What will occur if we cast **`flo f = (flo) x`** (*i.e.* try to represent the value stored in x as a `flo`)? Note: from a previous problem **`signedchar x = 0b10101000`** = -88.

Rounding   Underflow   Overflow   None of these

$-88 = -(64 + 16 + 8) = -(2^6 + 2^4 + 2^3) = -1011000_2 = -1.0110 \times 2^6$
Mantissa fits, but max exponent is 0b110 – bias = 6 – 3 = 3.

Floating Point Mathematical Properties

- Not <u>associative</u>: $(2 + 2^{50}) - 2^{50} \neq 2 + (2^{50} - 2^{50})$
- Not <u>distributive</u>: $100 \times (0.1 + 0.2) \neq 100 \times 0.1 + 100 \times 0.2$
- Not <u>cumulative</u>: $2^{25} + 1 + 1 + 1 + 1 \neq 2^{25} + 4$

<u>Exercises</u>:

6) Based on floating point representation, explain why each of the three statements above occurs.

Associative: Only 23 bits of mantissa, so $2 + 2^{50} = 2^{50}$ (2 gets rounded off). So LHS = 0, RHS = 2.

Distributive: 0.1 and 0.2 have infinite representations in binary point ($0.2 = 0.\overline{0011}_2$), so the LHS and RHS suffer from different amounts of rounding (try it!).

Cumulative: 1 is 25 powers of 2 away from 2^{25}, so $2^{25} + 1 = 2^{25}$, but 4 is 23 powers of 2 away from 2^{25}, so it doesn't get rounded off.

7) If `x` and `y` are variable type `float`, give two *different* reasons why `(x+2*y)-y==x+y` might evaluate to false.

(1) Rounding error: like what is seen in the examples above.
(2) Overflow: if `x` and `y` are large enough, then `x+2*y` may result in infinity when `x+y` does not.