Data Representation: Bits, Bytes, Ints and Floats

Aaina Arun, 1/23/22 (some activities developed by Prof. S. Garcia, UCSD)

Conversions: Decimal Binary Hexadecimal

Convert 0100 1000 1011 0101
to Hex:
to Decimal:

Convert 0x18213
to Decimal:
to Binary I

p.s: this table is good to have on your exam cheatsheet!

0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	A	10	1010
3	3	0011	B	11	1011
4	4	0100	C	12	1100
5	5	0101	D	13	1101
6	6	0110	E	14	1110
7	7	0111	F	15	1111

Boolean Algebra:

1001 1010 & 0000 0000	1001 1010 | 0000 0000	1001 1010 ^ 1111 1111
1001 1010 & 1111 1111	1001 1010 | 1111 1111	1001 1010 ^ 1001 1010

Encoding Ints: Unsigned and Signed:

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

$$B2T(X) = \underbrace{-x_{w-1} \cdot 2^{w-1}}_{\text{SIGN BIT}} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Assuming you only have 4 bits to represent a 2's complement int, fill in the following table:

x	x in binary	-x in binary
1		
2		
7		
-8		

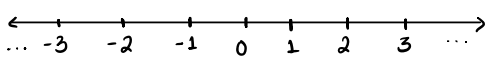
Generally, you can negate (find the additive inverse of) a number by taking its complement and then adding 1

What's the exception?

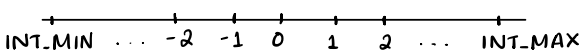
Integers vs. Ints:

The integers are an infinite set, but we only have a finite amount of resources to represent them.

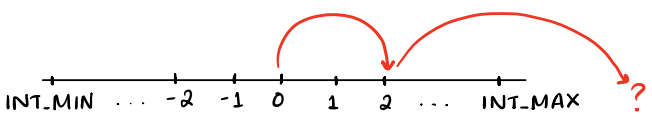
Mathematicians get to use a number line that stretches infinitely in both directions...



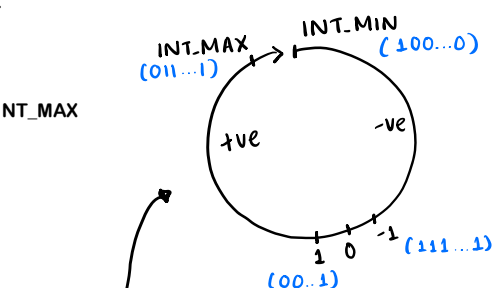
But programmers are limited to a fixed range. Let's call the endpoints INT_MIN and INT_MAX



But now, how do we handle operations that give us numbers out of our range?



We let our number line wrap around itself!

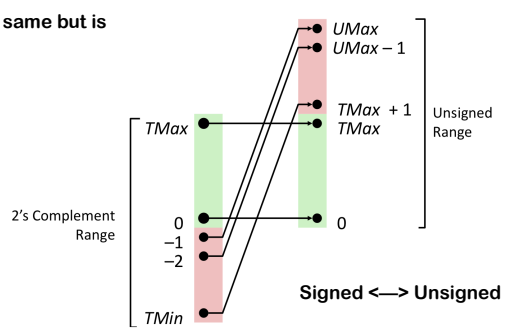


Casting:

When casting between signed and unsigned, the bit pattern stays the same but is reinterpreted.
When casting between shorts/ints/longs:
- expanding: sign extension
- truncating: simply drop higher order bits

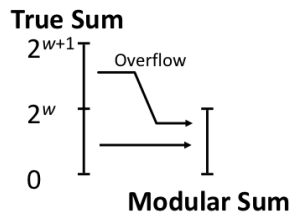
```
unsigned a = 0;
int b = 1;
```

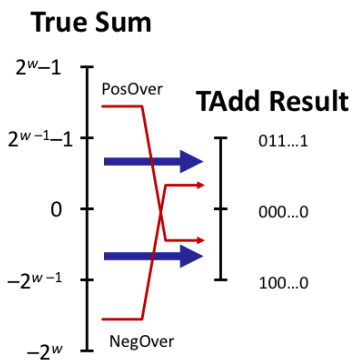
- What is the result of (a-b)?



Addition and Multiplication: Modular Arithmetic

Does bitwise addition and then drops any carry bits that don't fit.





This will be helpful on DataLab!

When overflow happens, we end up wrapping around our number circle. Convince yourself that we only wrap around at most once:

Given that we only wrap around at most once, describe how we can detect if an overflow has occurred when adding two numbers:

Left shifts can do power-of-two multiplication.
Right shifts can (almost) do power-of-two division. What's the issue with using right shifts to division with negative numbers?

Activity: BitCount

Let's count how many bits are set in a number. You can use any operator allowed in DataLab. Return the number of bits that are set (I.e., the number of 1 bits)

Using 1 op, you can return the BitCount of a 1-bit number. How?

How about if there are two bits in the input? (4 ops max)

```
int bitCount2Bit( int x ){
      int bit1 = _________________;
      int bit2 = _________________;
      return ____________ + ___________;
}
```

How about four bits? (8 ops max)

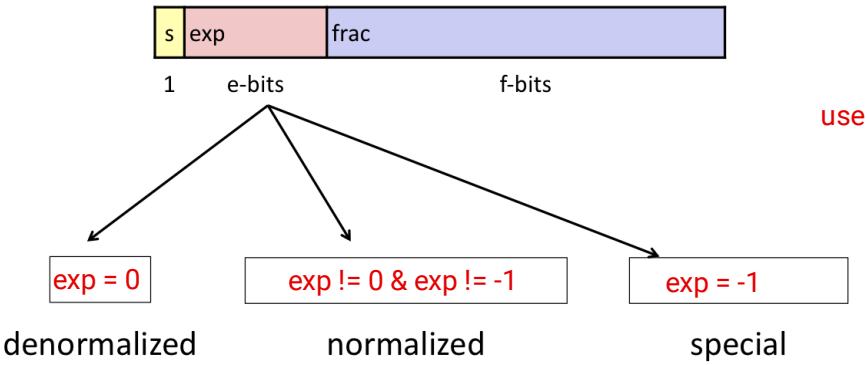
```
int bitCount4Bit( int x ){
      int mask = _________________;
      int halfSum = _________________;
      int mask2 = _________________;
      return ____________ + ____________;
}
```

How about if there are 8 bits? (12 ops max)

```
int bitCount8Bit( int x ){
      int mask = _________________;
      int quarterSum = _________________;
      int mask2 = _________________;
      int halfSum = _________________;
      int mask3 = _________________;
      return ____________ + _____________;
```

Reals vs. Floats:

The real numbers are an uncountably infinite set, so we can't truly encode them. We use floats as an approximation. They come in three flavours:



$$v = (-1)^s M 2^E$$

use unsigned exp value for E calculation!

How is E calculated for a normalised value?

E = exp - bias

How is E calculated for a denormalised value?

E = 1 - bias

- **Exponent is encoded as a biased value:**
  - **Bias = 2^(k-1) -1, where k is the number of exponent bits**
  - **How is**
  - **this way you can store exp as an unsigned value instead of 2's complement**
  - **if exp > bias, the exponent is positive, if exp < bias, the exponent is negative, and if exp = bias, the exponent is 0**
- **Why is biasing better than just storing the exp as a two's complement number?**

- On "int" number lines, off-balancing b/w positive and negative side of number line is only a single number. But when number is in exponent scientific notation, the off-balancing affects many numbers, each fraction raised to that power.
- Arithmetic would be challenging since the logical num line would not be contiguous
- Would make comparisons of exponents more complex b/c negatives appear larger than positives.