Computer Science 324
Computer Architecture
Mount Holyoke College
Fall 2009

Topic Notes: Bits and Bytes and Numbers

Number Systems

This is review, given the 221 prerequisite, so we will just go through it quickly.

Question: how high can you count on one finger?

That finger can either be up or down, so you can count 0, 1 and that's it.

(Computer scientists always start counting at 0, so you should get used to that...if you aren't already.)

So then... How high can you count on one hand/five fingers?

When kids count on their fingers, they can get up to 5. The number is represented by the number of fingers they have up.

But we have multiple ways to represent some of the numbers this way: 1 0, 5 1's, 10 2's, 10 3's, 5 4's and 1 5.

We can do better. We have 32 different combinations of fingers up or down, so we can use them to represent 32 different numbers.

Given that, how high can you count on ten fingers?

To make this work, we need to figure out which patterns of fingers up and down correspond to which numbers.

To keep things manageable, we'll assume we're working with 4 digits (hey, aren't fingers called digits too?) each of which can be a 0 or a 1. We should be able to represent 16 numbers. As computer scientists, we'll represent numbers from 0 to 15.

Our 16 patterns are base 2, or *binary*, numbers. In this case, we call the digits *bits* (short for **b**inary dig**its**).

Each bit may be 0 or 1. That's all we have.

Just like in base 10 (or *decimal*), where we have the 1's (10^0) place, the 10's (10^1) place, the 100's (10^2) place, etc, here we have the 1's (2^0), 2's (2^1), 4's (2^2), 8's (2^3), etc.

As you might imagine, binary representations require a lot of bits as we start to represent larger values. Since we will often find it convenient to think of our binary values in 4-bit chunks, we will also tend to use base 16 (or *hexadecimal*).

Since we don't have enough numbers to represent the 16 unique digits required, we use the numbers 0–9 for the values 0–9 but then the letters A–F to represent the values 10–15.

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

Any number can be used as the base, but the common ones we'll use are base 2, base 8 (*octal*, 3 binary digits), base 10, and base 16.

Since we will do so much work in binary, you will come to learn the powers of 2 and sums of common powers of 2.

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536.

Numbers like 49152 are common also (16384+32768).

Also, you'll get to know the numbers $2^n - 1$. Why?

Number representations

Some terminology, most of which you've likely seen before.

1. bit – 0 $\equiv$ False, 1 $\equiv$ True

2. *byte* (also *octet*) – term coined in 1956

   See `http://www.google.com/search?hl=en&q=byte+1956`

   A byte is often expressed as 2 hex digits, start with dollar sign or "0x" to make it clear that it's hex and not decimal:
   FE_{16}$ = 0xFE = 254 = 1111 1110$_2$

3. *nibble*/*nybble* (also *semioctet*) – 4 bits – "half a byte"

4. *word* – "the amount that a machine can handle with ease"

   16 bits for most of our purposes, a.k.a. `short`

5. `int`

   This is a C-language concept more than a machine concept.

   The number of bits in an int can vary but usually is 4 bytes/32 bits.

6. `long` or “longword” – almost always 32 bits

   2^{32} = 4.3 billion possible values

   You’ve heard of “32-bit” machines, like the Intel x86. This means they operate primarily on 32-bit values. More on what this all means later.

7. `long long` – 64 bits, a.k.a. “quadword”

   2^{64} values. 1.84×10^{19}

8. VAX 128-bit value: “octaword”

   2^{128} values. 3.40×10^{38}

9. bit and byte significance

   When placing the bits within a byte, they are almost always arranged with the *most significant bit* (msb) on the left, *least significant bit* (lsb) on the right.

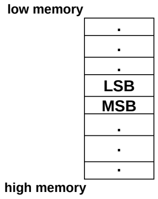
   Same idea for bytes making up words, longwords, etc.

0110 1010	0000 1100
MSB	**LSB**

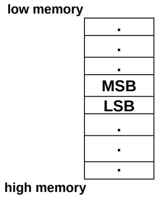
10. *endianness* – what order do we store these in, in memory

    As long we we’re consistent, it doesn’t really matter which way it is set up. No significant advantages or disadvantages.

    (a) *little endian* (x86)



    (b) *big endian* (Sun Sparc, 68K, PPC, IP “network byte order”)



An architecture can use either ordering internally, so long as it is consistent. However, endianness becomes important when we think about exchanging data among machines (networks). *Network byte ordering* (big endian) is required of networked data for consistency when exchanging data among machines that may use different internal representations.

The main architecture we'll be using, MIPS, is bi-endian. It can process data with either big or little endianness.

See Example:
`/home/jteresco/shared/cs324/examples/show_bytes`

Character representations

Computers only deal with numbers. Humans sometimes deal with letters. So we need to encode letters as numbers.

1. ASCII (1963) – American Standard Code for Information Interchange
   - (a) fits in a byte
   - (b) some you'll want to know:
     space (32 = 0x20)
     numbers ('0'-'9' = 48-57 = 0x30-0x39)
     lowercase letters ('a'-'z' = 97-122 = 0x61-0x7a), 96+letter pos
     uppercase letters ('A'-'Z' = 65-90 = 0x41-0x5a), 64+letter pos
   - (c) See: `man ascii`
2. Mild historical interest: EBCDIC (Extended Binary Coded Decimal Interchange Code) developed by IBM for punched cards in the early 1960s and IBM still uses it on mainframes today, as do some banking systems in some circumstances.
3. Unicode (1991), ASCII superset (for our purposes) – 2-byte characters to support international character sets

---

Memory model and pointers

We've talked about these bits and bytes and words, let's look at how these are organized in a computer's memory.

Aside: how much memory does your computer have? Your first computer? Your phone?

My first computer's memory was measured in kilobytes, my computer at home is measured in megabytes, computers in our department are measured in gigabytes, modern supercomputers can have terabytes.

Exponents:

1. K(ilo) = 2^{10} (2.4% more than 10^3)

2. M(ega) = 2^{20}
3. G(iga) = 2^{30}
4. T(era) = 2^{40}
5. P(eta) = 2^{50}
6. E(xa) = 2^{60}
7. Z(etta) = 2^{70}
8. Y(otta) = 2^{80} (21% more than 10^{24})

Rule of thumb: every $10^3 = 1000$ is approximately 2^{10} ($\log_2 10 \approx 3$).

Aside from aside: When you buy a hard drive, it's probably measuring gigabytes as billions of bytes not 2^{30} of bytes.

A simplistic but reasonably accurate view of a computer's memory:

The value of n determines how much memory you can have. Old systems: n=16 or n=24, modern systems: n=32, new systems: n=64 and these are becoming more common.

Think of this like a giant array of bytes.

We number these memory locations from 0 to $2^n - 1$, and we can refer to them by this number.

When we store a memory location in memory, we are storing a *pointer*.

The number of bits in a pointer determines how much memory can be addressed.

A pointer is just a binary value. If I have the value 0x10DE, I can think of that as referring to memory location $10DE.

Many modern systems let you access any byte, but this is not a requirement. The *addressable unit* may be a word or a longword.

In these systems, we can address a larger memory in the same number of bits in the pointer size, but can't get (directly) at every byte.

Even on a byte-addressable system, if we are treating a chunk of bytes as a word or longword, they may need to be *aligned* on 2-byte or 4-byte boundaries.

Unsigned Math

1. Unsigned addition – 4 bit numbers for simplicity

   Just like addition of base-10 numbers, really.

```
  0101     0101    1111
 +0011    +0111   +1110
```

```
-----    -----   -----
 1000     1100   11101
```

Ack! The answer sometimes doesn't fit! There's no place for that extra bit, so we've just added 15 and 14 and gotten 13 as our answer!

This is called an *overflow* condition and the extra bit in the 16's place is called a *carry out*.

2. Unsigned multiplication

   Sometimes the answer will fit, sometimes it won't.

```
  11     111
x100    x 11
----    ----
  00     111
 00     111
11      ----
----   10101
1100
```

Again, we have some overflow.

In many systems, the result of a multiplication of two n-bit numbers will be stored in a $2n$-bit number to avoid overflow.

Signed Math

So far, we've ignored the possibility of negative numbers.

How can we represent a signed integer?

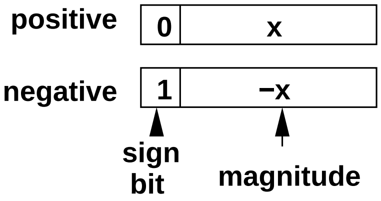
- Signed Magnitude

  The simplest way is to take one of our bits and say it's a sign.

  With n bits, we can now represent numbers from $-(2^{n-1} - 1)$ to $(2^{n-1} - 1)$

  Positive numbers just use the unsigned representation.

  Negative numbers use a 1 in the *sign bit* then store the magnitude of the value in the rest.

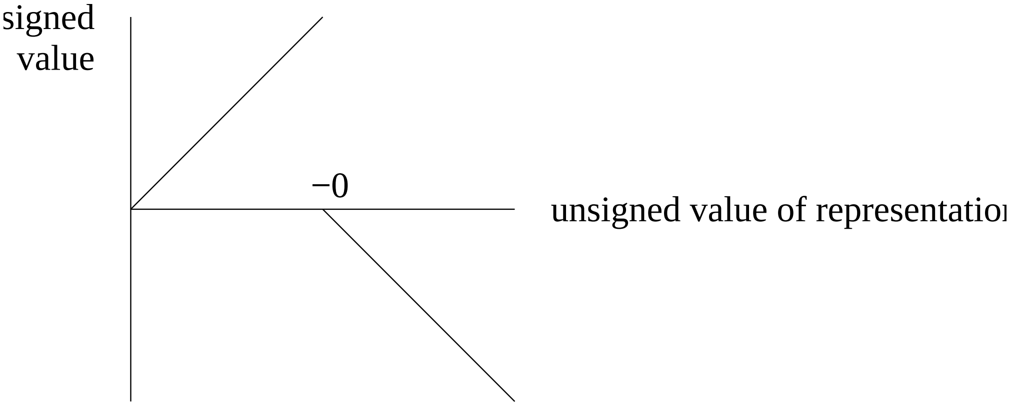


  Straightforward, makes some sense:

- You want to negate a value, you just switch its sign bit.
- You want to see if a value is negative, just look at the one bit

Note: two zeroes! +0 and -0 are distinct values.

Let's see how the values fall, also.



Disadvantage: direct comparison of two values differs between signed and unsigned values with the same representation. In fact, all negative numbers seem to be "bigger than" all positive numbers. Ordering of negatives is reverse of the ordering of positives.

- Excess N (actually used)

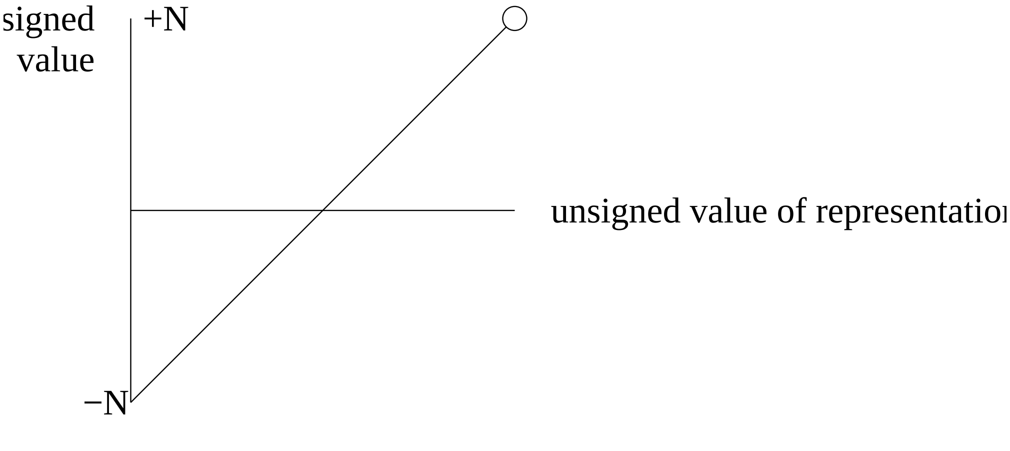
  Here, a value x is represented by the non-negative value x+N.

  With 4-bit numbers, it would be Excess 8

```
1000 = 0 (!)
0111 = -1
0000 = -8
1111 = 7
```

So our range of values is -8 to 7.

We eliminated the -0, plus the direct comparison works nicely.



- 1's complement

  This representation is "fairly frequently used in specific situations".

  For non-negative x, use the unsigned representation of x

  For negative x, use the **bit-wise complement** of -x

  C programming tip: the $\sim$ operator will do a bitwise complement.

  Examples:

  $$0 = 0000$$

  $$-1 = \overline{0001} = 1110$$

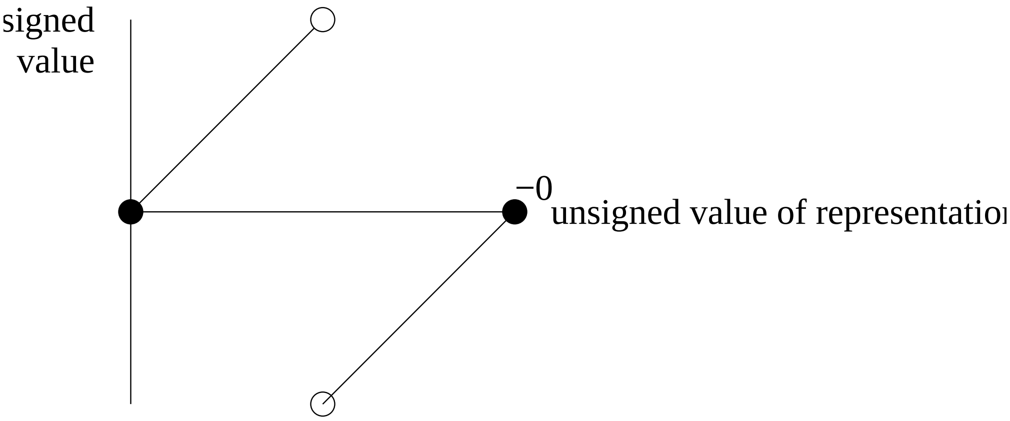
  $$-0 = \overline{0000} = 1111$$

  $$-7 = \overline{0111} = 1000$$

  Issues:

  – we have a -0

  – we can compare within a sign, but otherwise need to check sign



  Range: -7 to +7.

- 2's complement (the standard and default)

  For non-negative x, use the unsigned representation of x

  For negative x, use the complement of -x, then add 1 (weird!)

  $$0 = 0000$$

  $$-0 = \overline{0000}+1 = 1111+1 = 0000$$

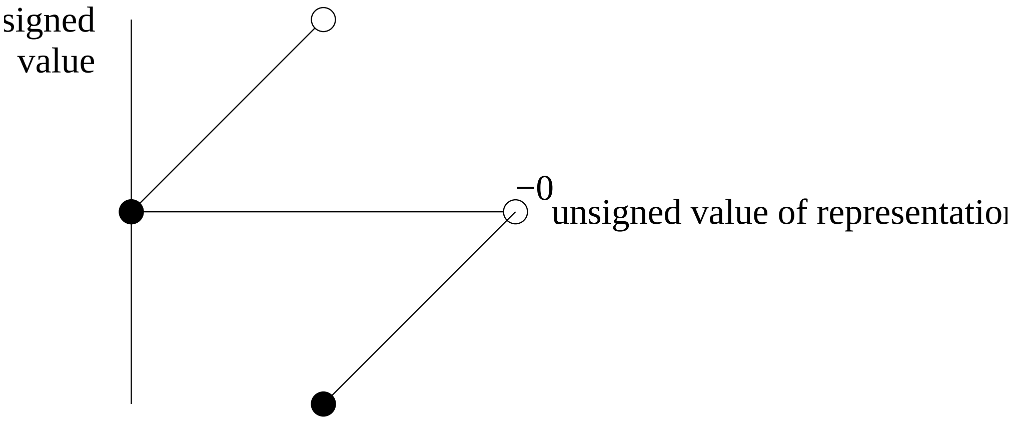
  Now, that's useful. 0 and -0 have the same representation, so there's really no -0.

$$1 = 0001$$

$$-1 = \overline{0001}+1 = 1110+1 = 1111$$

Also, very useful. We can quickly recognize -1 as it's the value with all 1 bits.

Another useful feature: 1's bit still determines odd/even (not true with 1's complement)



Like 1's complement, we can compare numbers with the same sign directly, otherwise have to check sign.

Note: Fortran had an if statement:

```
IF (I) GOTO 10,20,30
```

which translated to if I is negative, goto 10, if 0, goto 20, if positive, goto 30.

It is easy to check these cases with 2's complement.

All 4-bit 2's Complement numbers:

0000	= 0	1000	= -8
0001	= 1	1001	= -7
0010	= 2	1010	= -6
0011	= 3	1011	= -5
0100	= 4	1100	= -4
0101	= 5	1101	= -3
0110	= 6	1110	= -2
0111	= 7	1111	= -1

Note that the negation operation works both way. Take the 2's complement of a number then take the 2's complement again, you get the number back.

Signed addition

How does signed addition work with 2's complement?

```
  3    0011   -3    1101   4     0100   -4     1100   4     0100
 +4    0100   -4    1100   4     0100   -5     1011   5     0101
  ---------   ----------   ----------   -----------   ----------
  7 (0)0111   -7 (1)1001   8? (0)1000   -9? (1)0111   9? (0)1001
       OK            OK           -8 !          +7 !         -7 !
```

We were fine with 3+4.

To clarify:

- *carry out* is the extra 1 bit we generate that doesn't fit
- *overflow* is the condition where the answer is incorrect
- With unsigned addition, we have carry out iff we have overflow
- With signed addition, this is not the case:
  - (-3)+(-4) produces a carry out but no overflow (-7 is the right answer).
  - 4+4 and 4+5 do not produce a carry out, but produce overflow (-8 and -7 are the wrong answers)
  - (-4)+(-5) produces a carry out and overflows

How can we tell if we had a true overflow? If the carry in and carry out of the most significant bit are different, we have a problem.

Excellent feature: addition in 2's complement is same as unsigned, so if we have a circuit that adds unsigned numbers, it works for 2's complement also!

Subtraction is done by negating the subtrahend and adding.

What about signed addition for 1's complement?

```
  1  0001   -4  1011   -0  1111
 +4  0100   +4  0100   +1  0001
---------   --------   --------
  5  0101   -0  1111      10000
```

So -0+1 is 0. Not good. We need to add back in the carry out, to get 0001=1. We add in non-zero carry-outs until there is a 0 carry out.

You will practice these in the lecture assignment due Friday.

Multiplication

Can we use the same approach we used for unsigned?

```
  -1  1111
  x3  0011
  --------
      1111
     1111
  --------
    101101
```

If we take the low 4 bits, we have -3, as we should.

But if we're mutliplying 2 4-bit numbers and expecting an 8-bit result (reasonable thing to do), we don't have the right answer. 00101101=45.

We need to "sign extend" the numbers we're multiplying to 8 bits first:

```
  -1  11111111
  x3  00000011
  ------------
      11111111
     11111111
  ------------
    1011111101
```

Now, we truncate off the bits that didn't fit and we have -3.

Something that wouldn't fit in 4 bits?

```
  4  00000100
x-6  11111010
-------------
            0
         100
       100
      100
     100
    100
   100
-------------
   1111101000 = -24 (good)
```

Or two negatives that won't fit?

```
 -5     11111011
x-3     11111101
```

```
----------------
        11111011
      11111011
     11111011
    11111011
   11111011
  11111011
 11111011
----------------
        00001111 = 15 (good)
```

There's a bunch of other stuff before that, but it all falls out.

General division? Not something we're going to be concerned about.

Multiplying (and dividing?) by powers of 2: use a bit shift!

Does this always work?

To multiply 3x4, take 0011, shift left by 2, get 1100 (12).

To divide 27 by 8, take 00011011, shift right by 3, get 00000011 (3).

We lose the fraction, but these are ints, so of course we do.

This is fast! Computers like multiplying and dividing by powers of 2.

Question: Is adding 1 any simpler than adding an arbitrary n?

Logical operations

Our final topic on bits and numbers deals with logical operations on data.

Typically, we think of 1 as "true", 0 as "false" but in many circumstances, any non-zero value may be considered "true".

C and other high-level languages have logical operators that return 0/1 values:

(==, `!`, `!=`, &&, `||`)

You've used these in your Java or other programming.

Side note: recall the idea of *short-circuit evaluation*.

There are also bitwise logical operators that work on all bits in their operands:

- bitwise AND (&) – result true when both operands true

&	0	1
0	0	0
1	0	1

- bitwise OR (`|`) – result true when either operand true

|	0	1
0	0	1
1	1	1

- bitwise XOR (ˆ) – result true when exactly one operand true

ˆ	0	1
0	0	1
1	1	0

- bitwise complement (˜) – result true when operand false

~	
0	1
1	0

- Addition (+) *vs.* bitwise XOR (ˆ).

+	0	1
0	00	01
1	01	10

  Note: 2's bit is a logical and, 1's bit is an XOR.

  Remember this for later.

- Bitwise shift by n (`x>>n` or `x<<n`)

  "logical shift right" and "logical shift left"

  `i>>1` is division by 2

  For negatives, a logical shift right gives the wrong answer:

  (-2) >> 1 would take 1110 and give 0111, which is 7. Definitely not -2/2.

  The arithmetic shift copies in the sign bit to the values being shifted in.

  So (-2) >> 1 would take 1110 and get 1111, the right answer (-1)

  In C, >> is a logical shift right for an unsigned value, an arithmetic shift right for a signed value (which, fortunately, is exactly the right thing to do).

  In Java (but not C) arithmetic shift right (>>>), but there is no arithmetic shift left.

  `j<<k` is multiplication by 2^k

  (Aside: C has no power operator – call pow() – *don't* use ˆ (!))

- Some interesting and useful operations

  - To set bit i in a value n (note bits are numbered right to left)

    ```
    n = n | (1 << i)
    ```

    or better yet

    ```
    n |= (1 << i)
    ```

- To mask bit i in value n (we want the value of bit i to remain unchanged, all others become 0)

  ```
  n &= (1 << i)
  ```

- To toggle bit i in n

  ```
  n ˆ= (1 << i)
  ```

- We can generalize this to any set of bits, e.g. to mask the low nibble:

  ```
  n &= 0xF
  ```

See Example:
`/home/jteresco/shared/cs324/examples/shiftyproduct`

Floating point values

So far we have ignored non-integer numbers. We can store any integer in our unsigned or signed formats, given enough bits.

What about all those other numbers that aren't integers? Rational numbers, or even real numbers?

Let's think about the way we represent these things in our "normal" world.

$$3.5, \frac{2}{3}, 1.7 \times 10^{14}$$

We can use decimal notation, fractions, scientific notation.

Fractions seem unlikely as our binary representation, but we can use the decimal notation (actually, instead of a decimal point, we have a *radix* point).

11.1 = 2+1+$\frac{1}{2}$ = 3.5, 0.11 = $\frac{1}{2}$+$\frac{1}{4}$ = $\frac{3}{4}$

Just like we can't represent some fractions in decimal notation, we can't represent some fractions in binary notation either.

Remember $\frac{1}{3} = .\overline{3}$

Consider: $.1\overline{01}$

What value is this? $\frac{1}{2} + \frac{1}{8} + \frac{1}{32} + ...$

$$\begin{aligned} x &= .1\overline{01} \\ \frac{1}{2}x &= .0\overline{10} \\ x + \frac{1}{2}x &= .\overline{1} = 1 \\ x &= \frac{2}{3} \end{aligned}$$

How about $.\overline{1100}$?

$$\begin{aligned} x &= .\overline{1100} \\ \frac{1}{4}x &= .\overline{0011} \\ x + \frac{1}{4}x &= 1 \\ x &= \frac{4}{5} \end{aligned}$$

How can we denote $\frac{1}{5}$?

1. Multiply by 2, write integer part.
2. Keep fractional part, repeat until 0.
3. $\frac{1}{5} = .001100110011...$

Lots of decisions about how we place the radix point, etc. We want to store a wide range of values, but we're limited to 2^n unique values in any n-bit representation.

Scientific notation helps us here. Consider some examples:

$$\begin{aligned} .0001011 &= 1.011 \times 2^{-4} \\ .1 &= 1. \times 2^{-1} \\ 1.1 &= 1.1 \times 2^{0} \\ -101 &= -1.01 \times 2^{2} \\ 1111 &= 1.111 \times 2^{3} \end{aligned}$$

Floating point = integer part $+$ mantissa $\times$ 2^{exponent}

sign	**mantissa**	**exponent**

If we use binary version of scientific notation, we note that all numbers (other than 0) have a leading 1. Needn't store it! The *phantom 1* bit.

Mantissa is fractional, with MSb the $\frac{1}{2}$'s bit, etc.

Exponent stored in excess notation (helpful for *relatively* dumb hardware that must align fractional values before addition or subtraction).

What about 0? That should be the all-0's value. That really would represent something like 1.0×2^{-127}.

Trying to store something smaller than that value would result in a *floating point underflow*.

There are many standards, which can be hard to implement. These will include several useful and unusual values, such as $+\infty$, $-\infty$, `NaN` (not a number), etc.

You saw some of this in 221 but we will largely ignore floating point details for this course. You can look up the details if and when you want or need to know.