SIENAcollege
Computer Science

Computer Science 220
Assembly Language & Comp. Architecture
Siena College
Fall 2010

Topic Notes: Bits and Bytes and Numbers

Binary Basics

At least some of this will be review, but we will go over it for completeness.

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 ≡ False, 1 ≡ 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 number of bits depends on the architecture, but we might think of it as "the amount that a machine can handle with ease".

   16 bits or 32 bit for most of our purposes.

Note: a 16-bit word may be referred to as 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" – this almost always is 32 bits

   We have 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}

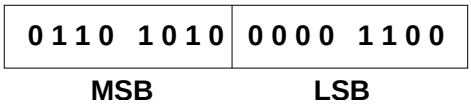
8. VAX 128-bit value: "octaword"

   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.

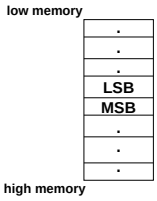
   Same idea for bytes making up words, longwords, etc.



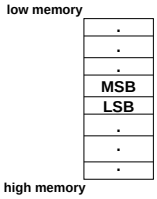
10. *endianness* – what order do we store these in, in memory

    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:
`~jteresco/shared/cs220/examples/show_bytes`

Character representations

Computers only deal with numbers. Humans sometimes deal with letters. So we just need agree on how 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. Of 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, lower-end computers might still be measured in megabytes, most current computers 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.

We will consider a simplistic but reasonably accurate view of a computer's memory: just think of it as a big table of locations.

The value of n determines how much memory you can have. Old systems: n=16 or n=24, many 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 we cannot (directly) access each individual 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

We next consider how to do arithmetic on binary numbers.

We begin with *unsigned* addition, where all of the values are nonnegative integers, and we work with 4-bit numbers for simplicity.

This works just like addition of base-10 numbers, if you can recall that procedure.

```
 0101     0101    1111
+0011    +0111   +1110
-----    -----   -----
 1000     1100   11101
```

The first couple were straightforward, but then we encounter a problem... 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*.

Unsigned multiplication

Again, sometimes the answer will fit, sometimes it won't. But the chances of a problem are much greater.

```
   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

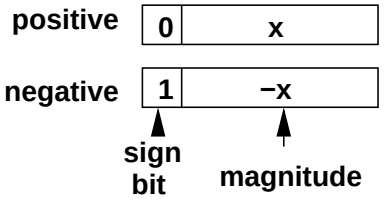
  The simplest way is to take one of our bits and say it's a sign: a 0 means positive, and a 1 means negative.

  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.

Typically the sign bit is the highest-order bit.

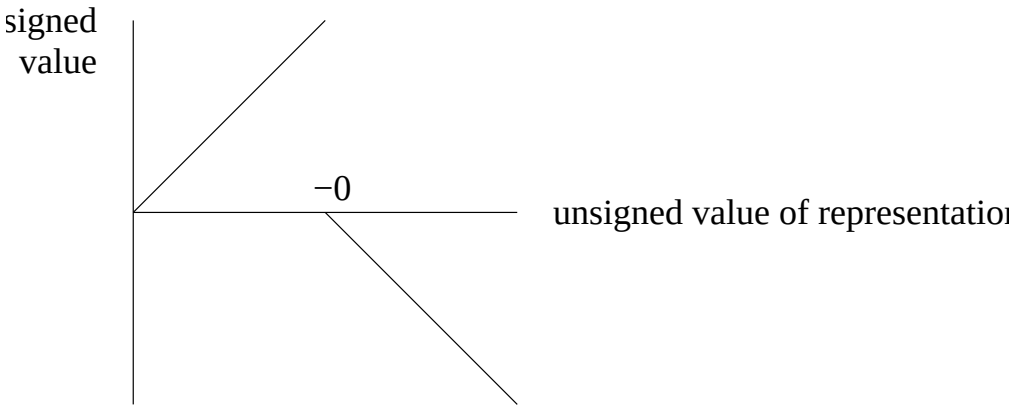


This idea is very straightforward, and makes some sense in practice:

- 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

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

Another property of signed binary representations that we will want to consider is how how the signed values fall, as a function of their unsigned representations.



So we do have a disadvantage: a direct comparison of two values differs between signed and unsigned values with the same representation. In fact, all negative numbers look to be "bigger than" all positive numbers. Plus, the ordering of negatives is reverse of the ordering of positives. This might complicate hardware that would need to deal with these values.

- Excess N

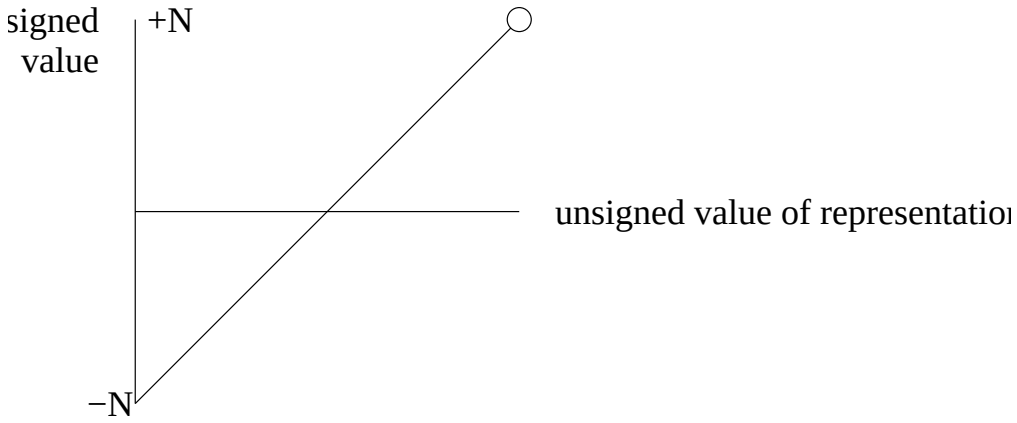
  Here, a value x is represented by the non-negative value x+N.

  With 4-bit numbers, it would make sense to use Excess 8, so we have about the same number of negative and positive representable values.

```
1000 = 0 (0 is not the all 0's pattern!)
0111 = -1
0000 = -8
1111 = 7
```

So we can represent a range of values is -8 to 7.

We eliminated the -0 problem, plus a direct comparison works the same as it would for unsigned representations.



Excess N representations are used in some circumstances, but are fairly rare.

- 1's complement

  For non-negative x, we just use the unsigned representation of x.

  For negative x, use the *bit-wise complement* (flip each bit) of $-x$.

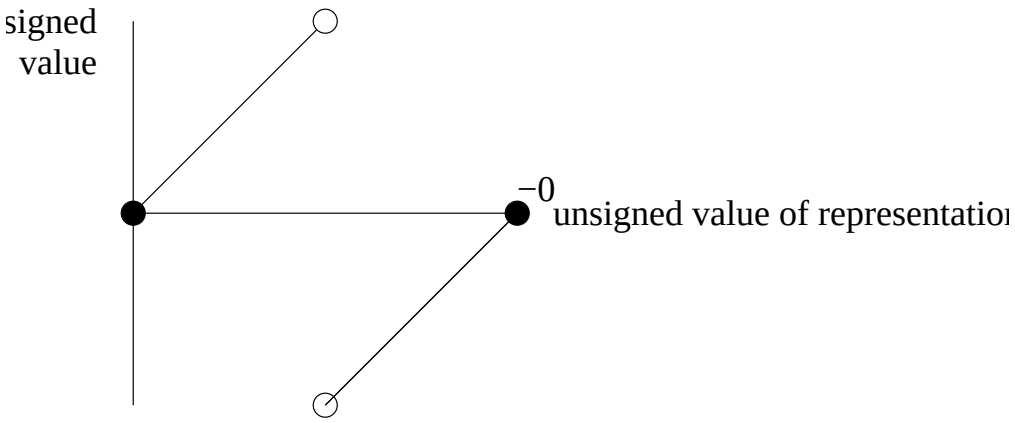
  C programming tip: the $\sim$ operator will do a bitwise complement.

  Examples:

```
 0 = 0000
      ____
-1 = 0001 = 1110
      ____
-0 = 0000 = 1111
      ____
-7 = 0111 = 1000
```

  Issues:

  - we have a -0
  - we can compare within a sign, but otherwise need to check sign



Range: -7 to +7.

Like Excess N, 1's complement is used in practice, but only in specific situations.

- 2's complement

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

  For negative x, use the complement of -x, then add 1 (that seems weird..).

```
 0 = 0000

-0 = 0000+1 = 1111+1 = 0000
```

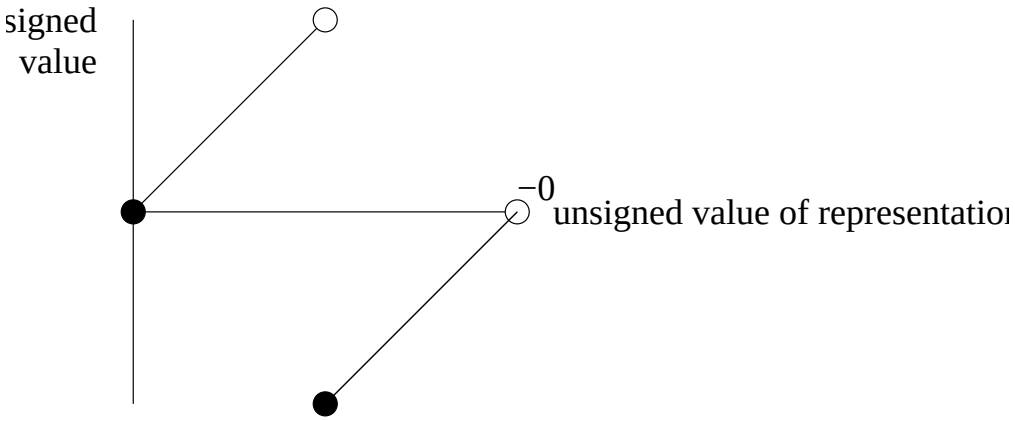
  Now, that's useful. 0 and -0 have the same representation, so there's really no -0.

```
 1 = 0001

-1 = 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.

Given these convenient properties, 2's complement representations are the standard and default unless there's some specific situation that calls for another representation.

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.

Here are all of the 4-bit 2's Complement numbers, which will become very familiar.

```
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
```

Notice that the negation operation works both ways: if you take the 2's complement of a number then take the 2's complement again, you get the original 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 !
```

Things are fine with 3+4. This makes sense since we are adding two positive numbers and get as a result a positive number that can be represented in 4-bit 2's complement.

However, this will not be the case all the time. Some terminology to get us ready.

- A *carry out* is when the carry bit we generate at the highest-order addition is a 1. It doesn't fit anywhere – we can't add it into the next place since there is no next place.
- An *overflow* is the condition where the answer is incorrect.

- With unsigned addition, it's easy. We have a carry out if and only if we have overflow.
- With signed addition, the situation is more complicated:
  - (-3)+(-4) produces a carry out but no overflow (and -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 overflow

So 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 overflow.

When developing a circuit to perform 2's complement addition, this is the rule we will use to detect overflow.

But note this excellent feature: other than the overflow detection, addition in 2's complement is same as unsigned addition, so if we have a circuit that adds unsigned numbers, it works for 2's complement also (once overflow detection is added)!

How can we do subtraction? Simple: negate the subtrahend and add.

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
```

2's Complement 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.

Note that we will throw away all places that don't fit in our 8-bit result, so why compute them?

How about when we have a product that doesn't fit in 4 bits?

```
  4  00000100
x-6  11111010
-------------
            0
          100
        100
       100
      100
     100
    100
   100
-------------
     11101000 = -24 (good)
```

Or two negatives that won't fit?

```
 -5     11111011
x-3     11111101
----------------
        11111011
      11111011
     11111011
    11111011
   11111011
  11111011
 11111011
----------------
        00001111 = 15 (good)
```

Division? Sure, we can do it, but we won't here.

Logical operations

Our next 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 high-level language programming, and should have a solid handle on them.

Side note: recall the idea of *short-circuit evaluation*. When do we know the result of

```
if (a && b && c && d)
```

or

```
if (a || b || c || d)
```

Well, for the logical AND, we know the whole expression is false as soon as we encounter any false. For the logical OR, we know the whole expression is true as soon as we encounter any true.

There is no need to continue evaluation once the boolean value of the overall expression is determined (so beware of *side effects* in your conditions!).

We will consider this idea more carefully when we work on assembly programming.

There are also *bitwise* logical operators that work individually 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 (`^`) – result true when exactly one operand true

^	0	1
0	0	1
1	1	0

- bitwise complement (`~`) – result true when operand false

~	
0	1
1	0

- Addition (+) – we have a 2-bit result

+	0	1
0	00	01
1	01	10

  Note that the 2's bit is just a logical AND, the 1's bit is an XOR.

  We'll use this later on when building adder circuits.

- 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 the right answer for $\frac{-2}{2}$.

  The *arithmetic shift* copies the sign bit for 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 ^ (that's the bitwise XOR!))

- 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 ^= (1 << i)
  ```
  – We can generalize this to any set of bits, e.g. to mask the low nibble:
  ```
  n &= 0xF
  ```

An example that uses these:

See Example:
`~jteresco/shared/cs220/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" base-10 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. More precisely, 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: $.\overline{10}$

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

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

How about $.\overline{1100}$?

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

$$x = \frac{4}{5}$$

How can we denote an arbitrary fractional value, say, $\frac{1}{5}$?

We can follow this procedure:

1. Multiply by 2, write integer part.
2. Keep fractional part, repeat until 0, or a repeating pattern emerges.

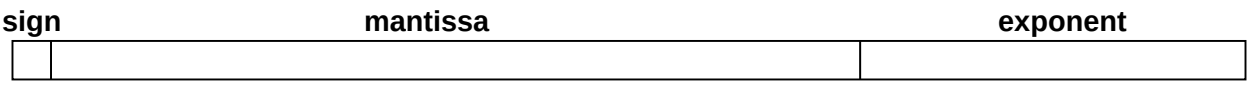
So $\frac{1}{5} = .001100110011...$

When representing these in the computer, we have lots of decisions to make, such as 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 $\times\ 2^{\text{exponent}}$



If we use binary version of scientific notation, we note that all numbers (other than 0) have a leading 1. So we need not store it! This is known as the *phantom 1* bit.

The *mantissa* is fractional, with the most significant bit representing the $\frac{1}{2}$'s bit, the next the $\frac{1}{4}$'s bit, etc.

The *exponent* is stored in excess notation (which is helpful for hardware that must align fractional values before addition or subtraction).

What about 0? It would be nice if that was the all-0's value. However, 00000000000000000000000000000000 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.

While it is important to have an idea of how floating-point representations work, we will largely ignore floating point details for this course. If you understand the basics, you can look up the details if and when you want or need to know.