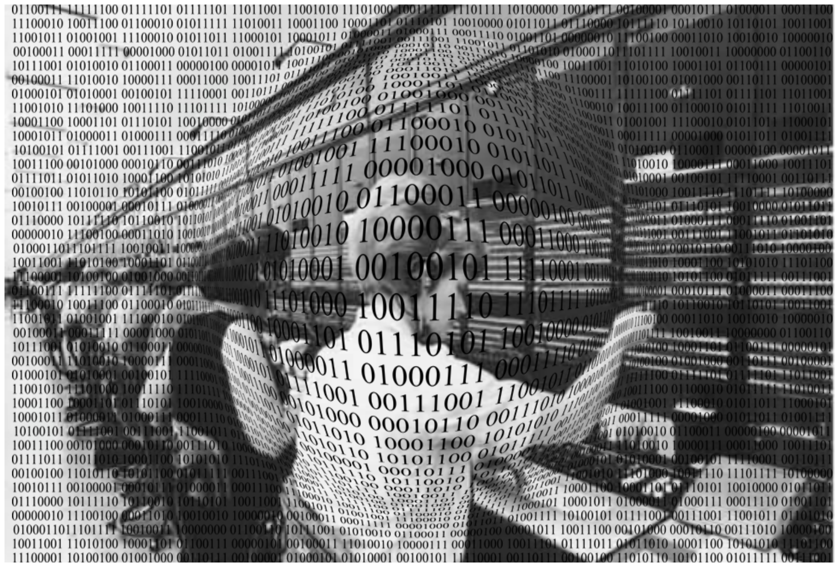
Binary Representations

"2 bits, 4 bits, 6 bits a byte!"



- Representing Information as bits
- Number Representations
- Other bits
- A quick review of Comp 211

Fixed-Size Codes

If all choices are equally likely (or we have no reason to expect otherwise), then a fixed-size code is often used. Such a code should use at least enough bits to represent the information content.

ex. Decimal digits 10 = {0,1,2,3,4,5,6,7,8,9}
4-bit BCD (binary coded decimal)
$\log_2(10/1) = 3.322 < 4$ bits

BCD		
0	-	0000
1	-	0001
2	-	0010
3	-	0011
4	-	0100
5	-	0101
6	-	0110
7	-	0111
8	-	1000
9	-	1001

ex. ~84 English characters = {A-Z (26), a-z (26), 0-9 (10), punctuation (8), math (9), financial (5)}

7-bit ASCII (American Standard Code for Information Interchange)
$\log_2(84/1) = 6.392 < 7$ bits

ASCII FOR TEXT

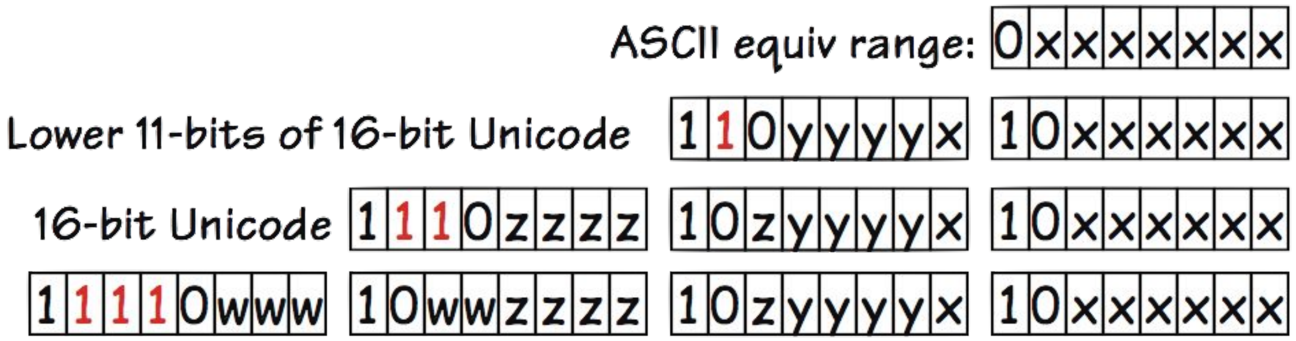
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
000	NUL	SOH	STX	ETX	EOT	ACK	ENQ	BEL	BS	HT	LF	VT	FF	CR	SO	SI
001	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
010		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
011	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
101	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
110	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
111	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

- For letters upper and lower case differ in the 6th "shift" bit
  10XXXXX is upper, and 11XXXXX is lower
- Special "control" characters set upper two bits to 00
  ex. cntl-g → bell, cntl-m → carriage return, cntl-[→ esc
- This is why bytes have 8-bits (ASCII + optional **parity**). Historically, there were computers built with 6-bit bytes, which required a special "shift" character to set case.

Unicode: A variable length text encoding

- ASCII is biased towards western languages. English in particular.
- There are, in fact, many more than 256 characters in common use:

  â, ö, ß, ñ, è, ¥, £, 揗, 敇, 휠, 力, ℵ, ℷ, ж, š
- Unicode is a worldwide standard that supports all languages, special characters, classic, extinct, and arcane.
- Several encoding variants 16-bit (UTF-8)
- Variable length (as determined by first byte)



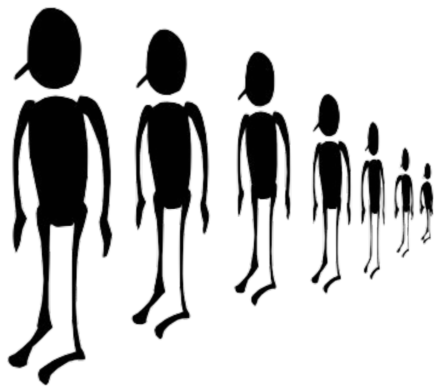
ASCII equiv range: 0xxxxxxx
Lower 11-bits of 16-bit Unicode 110yyyyx 10xxxxxx
16-bit Unicode 1110zzzz 10zyyyyx 10xxxxxx
11110www 10wwzzzz 10zyyyyx 10xxxxxx

Encoding Positive Integers

It is straightforward to encode positive integers as a sequence of bits. Each bit is assigned a weight. Ordered from right to left, these weights are increasing powers of 2. The value of an n-bit number encoded in this fashion is given by the following formula:

$$v = \sum_{i=0}^{n-1} 2^i b_i$$

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0



$$\begin{array}{rrr}
 & 2^1 = & 2 \\
+ & 2^2 = & 4 \\
+ & 2^5 = & 32 \\
+ & 2^6 = & 64 \\
+ & 2^7 = & 128 \\
+ & 2^8 = & 256 \\
+ & 2^9 = & 512 \\
+ & 2^{10} = & \underline{1024} \\
 & & 2022
\end{array}$$

Favorite Bits

- You are going to have to get accustomed to working in binary. Specifically for Comp 311, but it will be helpful throughout your career as a computer scientist.
- Here are some helpful guidelines:
  1. Memorize the first 10 powers of 2

$2^0 = 1$	$2^1 = 2$	$2^2 = 4$	$2^3 = 8$	$2^4 = 16$
$2^5 = 32$	$2^6 = 64$	$2^7 = 128$	$2^8 = 256$	$2^9 = 512$

  2. Memorize the prefixes for powers of 2 that are multiples of 10

2^{10} = Kilo (1024)	2^{40} = Tera (1024^4)
2^{20} = Mega (1024*1024)	2^{50} = Peta (1024^5)
2^{30} = Giga (1024*1024*1024)	2^{60} = Exa (1024^6)

Tricks with Bits

- The first thing that you'll do a lot of is ***clustering groups of contiguous bits.***
- Using the binary powers that are multiples of 10 we can do the most basic clustering.

  1. When you convert a binary number to decimal, first break it down from the right into clusters of 10 bits.
  2. Then compute the value of the leftmost remaining bits (1)
  3. Find the appropriate prefix (GIGA)
  4. Often this is sufficient (might need to round up)

$$01\boxed{0001100000}\boxed{0001100000}\boxed{0000101000}$$

A "Giga" something or other

Other Helpful Clusterings

Oftentimes we will find it convenient to cluster groups of bits together for a more compact written representation. Clustering by 3 bits is called **Octal**, and it is often indicated with a leading zero, **0**. Octal is not that common today.

$$v = \sum_{i=0}^{n-1} 8^i d_i$$

2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	1	1	1	0	1	0	0	0	0

$= 2000_{10}$

$$\underbrace{011}_{3}\,\underbrace{111}_{7}\,\underbrace{010}_{2}\,\underbrace{000}_{0}$$

03720

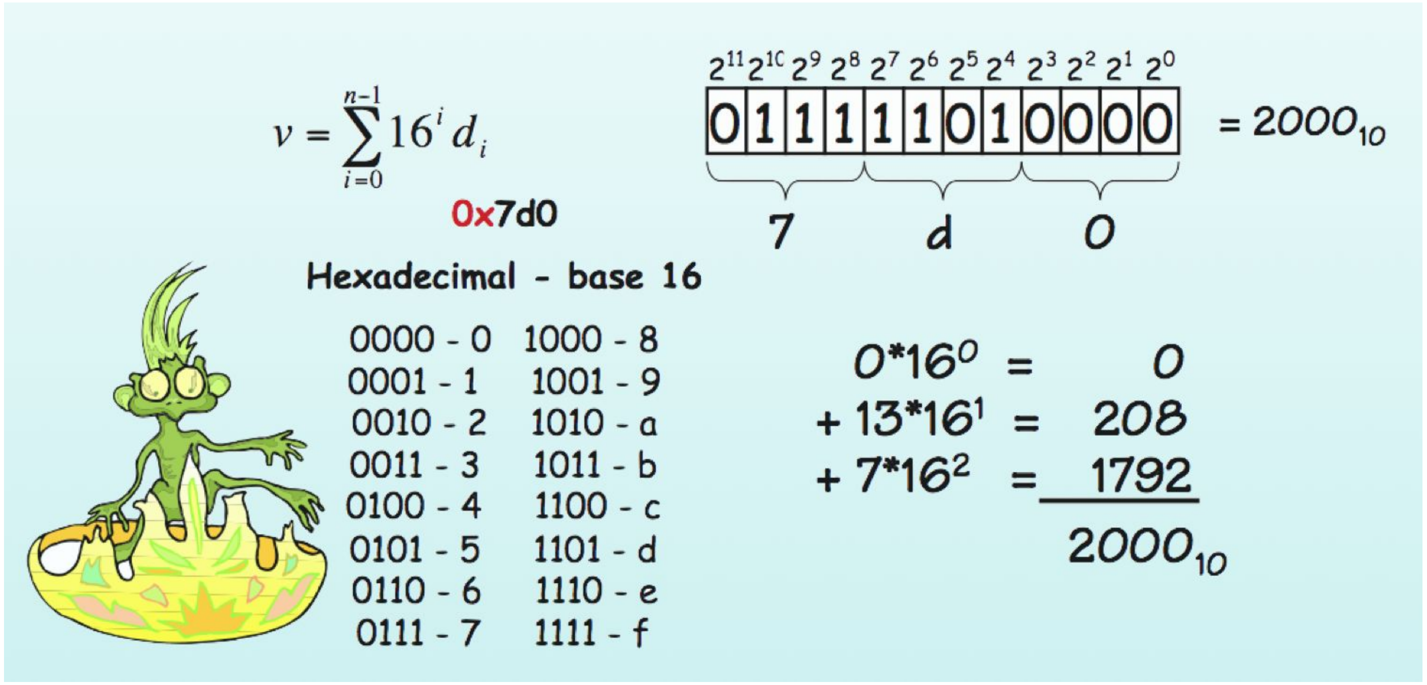
Octal - base 8

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

$$\begin{array}{rcr} 0*8^0 & = & 0 \\ +\ 2*8^1 & = & 16 \\ +\ 7*8^2 & = & 448 \\ +\ 3*8^3 & = & \underline{1536} \\ & & 2000_{10} \end{array}$$

One more Clustering

Clusters of 4 bits are used most frequently. This representation is called ***hexadecimal***. The hexadecimal digits include 0-9, and A-F, and each digit position represents a power of 16. Commonly indicated with a leading "0x".



$$v = \sum_{i=0}^{n-1} 16^i d_i$$

0x7d0

$$\begin{array}{cccccccccccc} 2^{11} & 2^{10} & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ 0&1&1&1&1&1&0&1&0&0&0&0 \end{array} = 2000_{10}$$

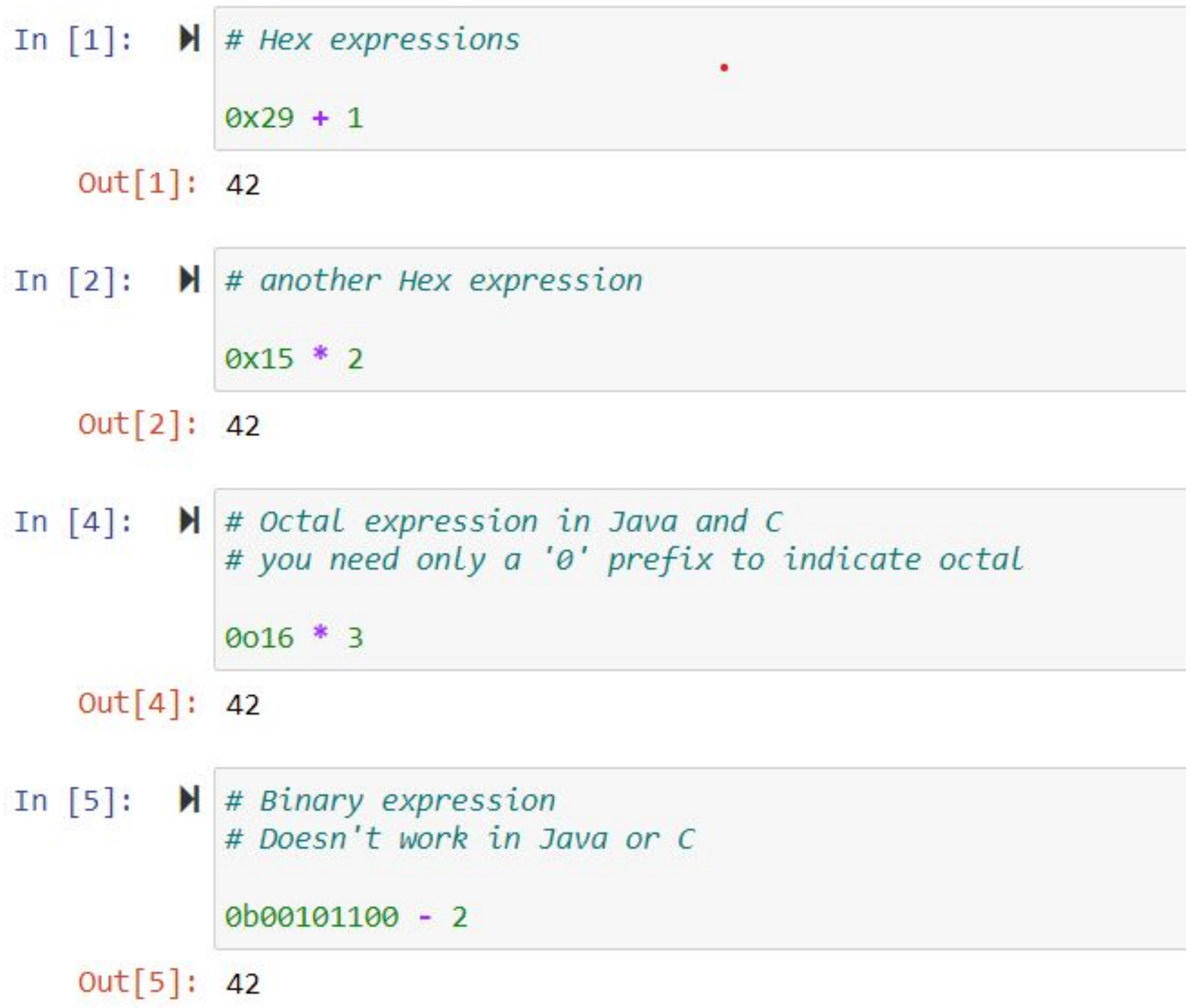
$$\underbrace{0111}_{7}\ \underbrace{1101}_{d}\ \underbrace{0000}_{0}$$

Hexadecimal - base 16

0000 - 0	1000 - 8
0001 - 1	1001 - 9
0010 - 2	1010 - a
0011 - 3	1011 - b
0100 - 4	1100 - c
0101 - 5	1101 - d
0110 - 6	1110 - e
0111 - 7	1111 - f

$$\begin{array}{rcr} 0*16^0 & = & 0 \\ +\ 13*16^1 & = & 208 \\ +\ 7*16^2 & = & \underline{1792} \\ & & 2000_{10} \end{array}$$

They've always been there...



```
In [1]:    # Hex expressions

           0x29 + 1

Out[1]:    42

In [2]:    # another Hex expression

           0x15 * 2

Out[2]:    42

In [4]:    # Octal expression in Java and C
           # you need only a '0' prefix to indicate octal

           0o16 * 3

Out[4]:    42

In [5]:    # Binary expression
           # Doesn't work in Java or C

           0b00101100 - 2

Out[5]:    42
```

Signed Integers

- Obvious method is to encode the sign of the integer using one bit.
- Conventionally, the most significant bit is used for the sign.
- This encoding of signed integers is called "SIGNED MAGNITUDE"

$$v = -1^S \sum_{i=0}^{n-2} 2^i b_i$$

S	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	0	0	1	1	1	1	1	1	0	0	1	1	0

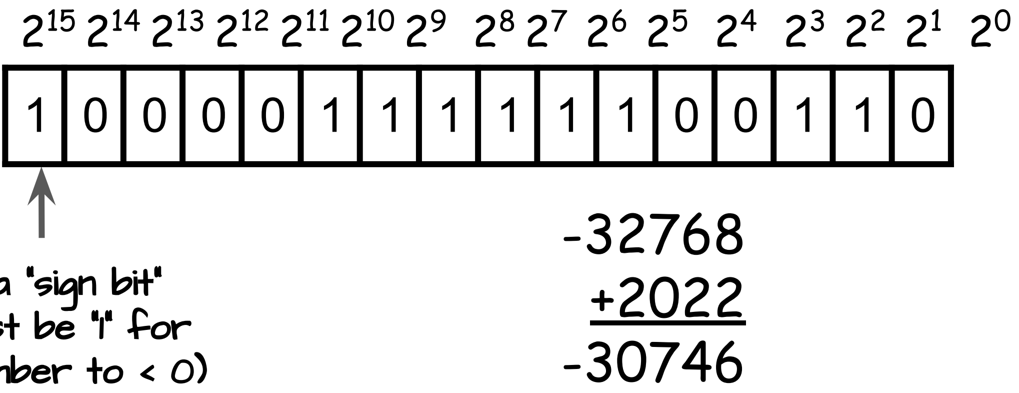
- 2022

- The Good
  - Easy to negate, easy to take absolute value
- The Bad
  - Two ways to represent "0", +0 and -0
  - Add/subtract is complicated; depends on the signs
- Not frequently used in practice
  - With one important exception that we'll discuss shortly

2's Complement Notation

- The 2's complement representation for signed integers is the most commonly used signed-integer representation.
- It is a simple modification of unsigned integers where the most significant bit is a negative power of 2.

$$v = -2^{n-1} b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i$$



- Huh?
  - Negative numbers seem hard to "read" (for humans)
  - Nonsymmetric range:

    For 16 bits the range is $-32768 \le x \le 32767$

Why 2's Complement?

- In the two's complement representation for signed integers, the same binary "addition procedure" (mod 2^n) works for adding any combination of positive and negative numbers.
- Don't need a separate "subtraction procedure" (carries only, no borrows)
- The "addition procedure" also handles unsigned numbers!
- In 2's complement adding is "adding" regardless of operand signs.
- You NEVER need to subtract when you use 2's-complement.
- Just form the 2's -complement of the subtrahend

$$\begin{array}{rcr} 55_{10} & = & 000000110111_2 \\ +10_{10} & = & 000000001010_2 \\ \hline 65_{10} & = & 000001000001_2 \end{array}$$

$$\begin{array}{rcr} 55_{10} & = & 000000110111_2 \\ +-10_{10} & = & 111111110110_2 \\ \hline 45_{10} & = & 1000000101101_2 \end{array}$$

Ignore this "carry"

2's Complement Tricks

- **Negation** - changing the sign of a number
  1. Invert every bit (i.e. 1 → 0, 0 → 1)
  2. Add 1

Example: $42_{10} = 000000101010_2$

$-42_{10} = 111111010101_2 + 1 = 111111010110_2$

- **Sign-Extension** - aligning different sized 2's complement integers
  - Simply copy the sign bit into higher positions

Example: 16-bit version of 42: $42_{10} = 0000000000101010_2$

16-bit version of -42: $-42_{10} = 1111111111010110_2$

Class Exercise

10's-complement Arithmetic (so you'll never need to borrow again)

Step 1) Write down two 3-digit numbers, where you'll subtract the second from the first

Step 2) Form the 9's-complement of each digit in the second number (the subtrahend)

Step 3) Add 1 to it (the subtrahend)

Step 4) Add this number to the first

Step 5) If your result is less than 1000, form the 9's complement of the sum, add 1, and remember your result is negative, otherwise subtract 1000

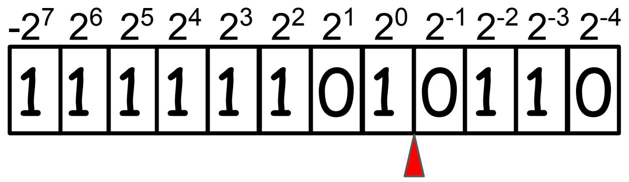
Helpful Table of the 9's complement for each digit

0 → 9	
1 → 8	
2 → 7	
3 → 6	
4 → 5	
5 → 4	
6 → 3	
7 → 2	
8 → 1	
9 → 0	

What did you get? Why weren't you taught to subtract this way?

Fixed-point numbers

- You can always assume that the boundary between 2 bits is a "binary point".
- If you **align** binary points between addends, there is no effect on how operations are performed.



$$11111101.0110 = -2^{7} + 2^{6} + 2^{5} + 2^{4} + 2^{3} + 2^{2} + 2^{0} + 2^{-2} + 2^{-3}$$
$$= -128 + 64 + 32 + 16 + 8 + 4 + 1 + 0.25 + 0.125$$
$$= -2.625$$

OR

$$11111101.0110 = -42 \times 2^{-4}$$
$$= -42 / 16$$
$$= -2.625$$

Repeated Binary Fractions

Not all fractions can be represented exactly using a **finite representation**. You've seen this before in decimal notation where the fraction 1/3 (among others) requires an infinite number of digits to represent (0.3333...).

In binary, a great many fractions that you've grown attached to require an infinite number of bits to represent exactly.

Example: $1/10 = 0.1_{10} = 0.0\overline{0011}\ldots_2 = 0.1\overline{9}\ldots_{16}$

$1/5 = 0.2_{10} = 0.\overline{0011}\ldots_2 = 0.\overline{3}\ldots_{16}$

$1/3 = 0.3_{10} = 0.\overline{01}\ldots_2 = 0.\overline{5}\ldots_{16}$

Finite Representations

- Computers use a **finite** set of bits (or certain fixed-sized bit clusters) to represent numbers.
- In fact, **everything** that a realizable computer does is limited by a finite set of bits.
- Through your mastery of mathematics, you've gradually grown used to infinite representations. So much so that finite representations seem odd
- One type of infinity that you've grown used to: **Infinite digits**

```
          ...0000000042.0000000000...
          ...00000000000.0000000000...001000
100000000...00000000000.0
```

- The concept an infinite supply of zero digits is conceptually elegant, but difficult to physically implement

Side Effects of being Finite

These examples assume a finite 16-bit representation

- You can "overflow"

$32767_{10} + 1_{10} = -32768_{10}$

```
  0111 1111 1111 1111₂
+ 0000 0000 0000 0001₂
  -------------------
  1000 0000 0000 0000₂
```

$-20000_{10} - 20000_{10} = 25536_{10}$

```
  1011 0001 1110 0000₂
+ 1011 0001 1110 0000₂
  -------------------
 10110 0011 1100 0000₂
```

- Certain numbers can't be negated

$-32768_{10} = -32768_{10}$

```
    1000 0000 0000 0000₂
    0111 1111 1111 1111₂
+   0000 0000 0000 0001₂
    -------------------
    1000 0000 0000 0000₂
```

Bias Notation

There is yet one more way to represent signed integers, which is surprisingly simple. It involves subtracting a fixed constant from a given unsigned number. This representation is called "Bias Notation".

$$v = \sum_{i=0}^{n-1} 2^i b_i - Bias$$

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	1	0	1	1	0

Example of Bias 127:

$$\begin{array}{lrr} & 9 \times 2^4 = & 144 \\ + & 6 \times 2^0 = & 6 \\ - & & 127 \\ \hline & & 23 \end{array}$$

Adding 2 numbers requires a subtraction to fix the result

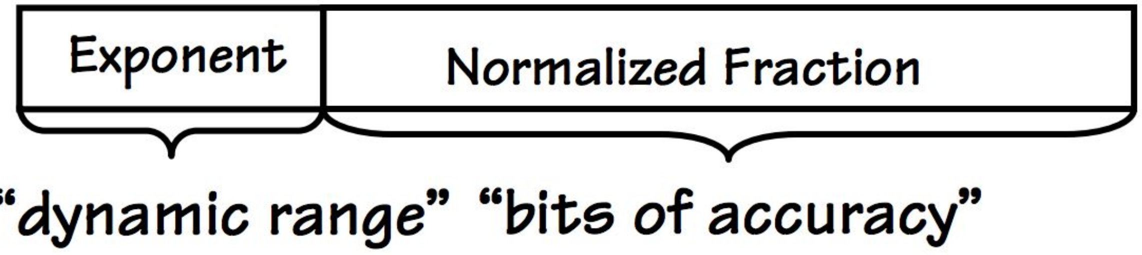
$$\begin{array}{lr} & 150 \\ + & 150 \\ - & 127 \\ \hline & 173 = 46 + 127 \end{array}$$

Why? Monotonicity when viewed as an unsigned number

Floating Point Numbers

Another way to represent numbers is to use a notation similar to **Scientific Notation**. This format can be used to represent numbers with fractions (3.90×10^{-4}), very small numbers (1.60×10^{-19}), and large numbers (6.02×10^{23}). This notation uses two fields to represent each number. The first part represents a normalized fraction (called the significand), and the second part represents the exponent (i.e. the position of the "floating" binary point).

$$\textit{Normalized}\quad \textit{Fraction} \times 2^{\textit{Exponent}}$$

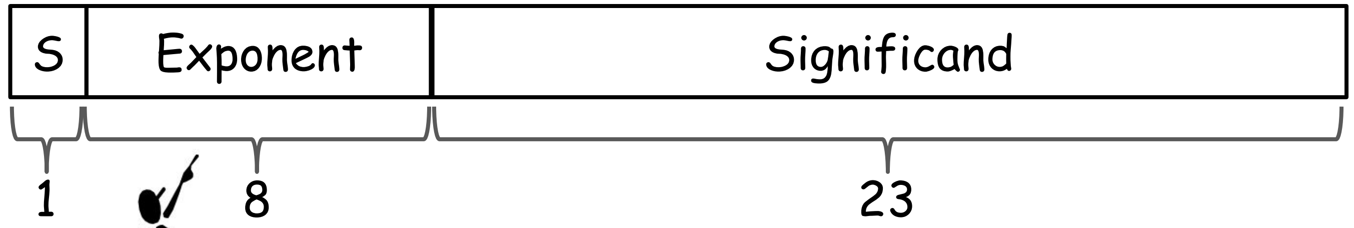


IEEE 754 Format

- Single precision format

This is effectively a signed magnitude fixed-point number with a "hidden" 1.

The 1 is hidden because it provides no information after the number is "normalized"



The exponent is represented in bias 127 notation. Why?

$$v = -1^{s} \times 1.\text{Significand} \times 2^{\text{Exponent}-127}$$

- Example: 52.25 = 00110100.01000000_2

Normalize: $001.10100010000000_2 \times 2^5$

(127+5)

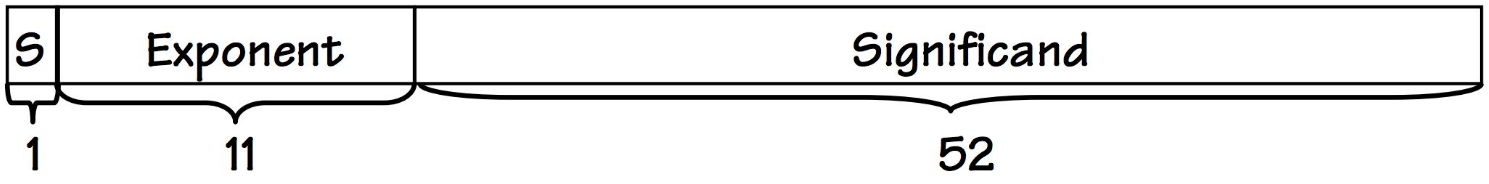
0 10000100 10100010000000000000000

0100 0010 0101 0001 0000 0000 0000 0000

52.25 = 0x42510000

IEEE 754 Limits and Features

- Single precision limitations
  - A little more than 7 decimal digits of precision
  - Minimum positive normalized value: ~1.18×10^{-38}
  - Maximum positive normalized value: ~3.4×10^{38}
- Inaccuracies become evident after multiple single precision operations
- Double precision format



$$v = -1^{s} \times 1.\text{Significand} \times 2^{\text{Exponent}-1023}$$

IEEE 754 Special Numbers

- Zero - ±0

  A floating point number is considered zero when its exponent and significand are both zero. This is an exception to our "hidden 1" normalization trick. There are also a positive and negative zeros.

S	000 0000 0	000 0000 0000 0000 0000 0000

- Infinity - ±∞

  A floating point number with a maximum exponent (all ones) is considered infinity which can also be positive or negative.

S	111 1111 1	000 0000 0000 0000 0000 0000

- Not a Number - NaN for ±0/±0, ±∞/±∞, log(-42), etc.

S	111 1111 1	non-zero

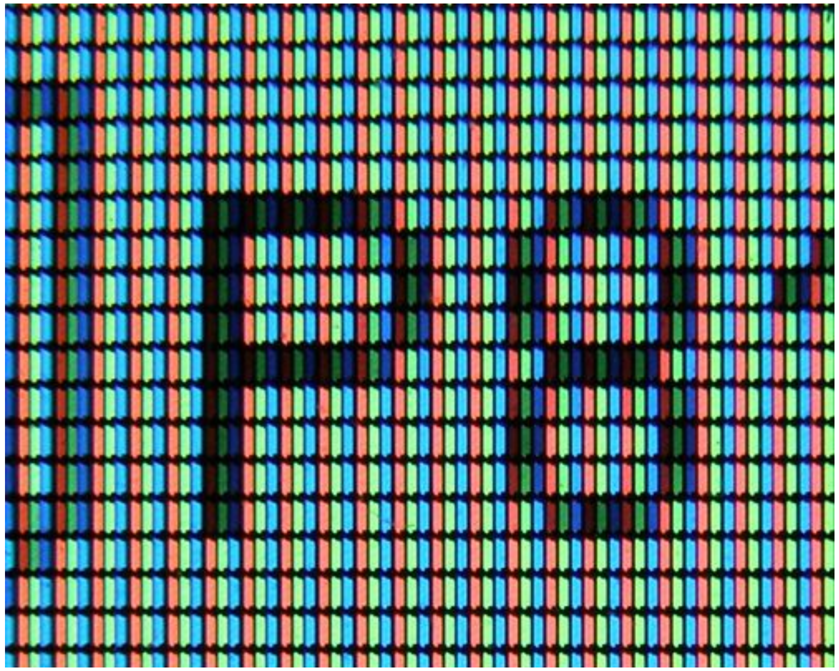
A Quick Wake-up exercise

What decimal value is represented by 0x3f800000, when interpreted as an IEEE 754 single precision floating point number?

Bits You can See

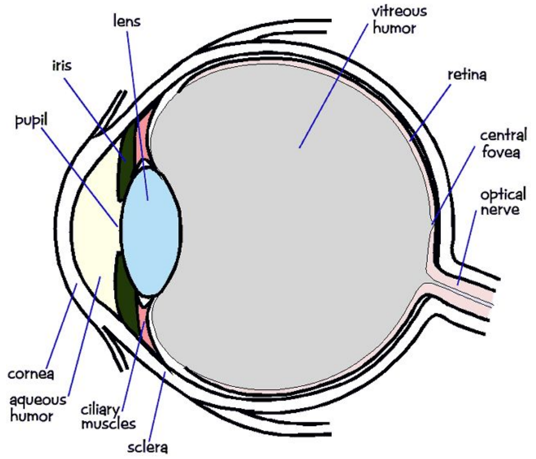
The smallest element of a visual display is called a "pixel". Pixels have three independent color components that generate most of the ***perceivable*** color range.

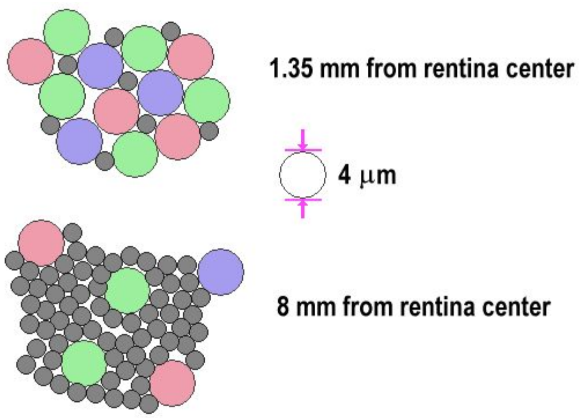
- Why three and what are they
- How are they represented in A computer?
- First, let's discuss this notion of perceivable



It starts with the Eye

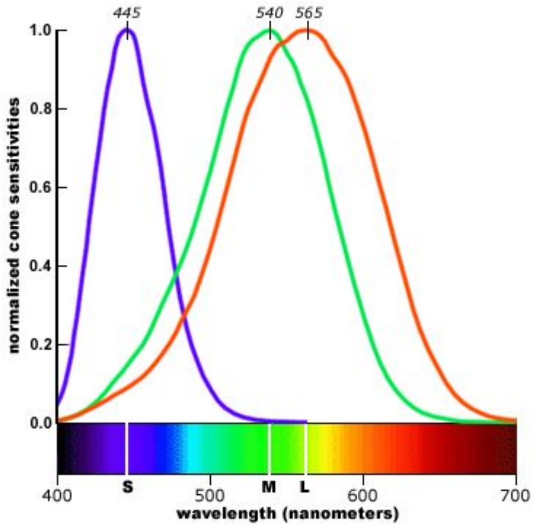
- The photosensitive part of the eye is called the retina.
- The retina is largely composed of two cell types, called rods and cones.
- Cones are responsible for color perception.
- Cones are most dense within the fovea.
- There are three types of cones, referred to as S, M, and L whose spectral sensitivity varies with wavelength.

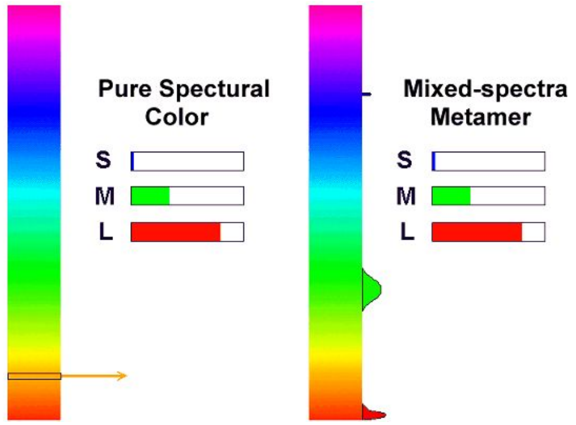




Why we see in color

- Pure spectral colors simulate all cones to some extent.
- Mixing multiple colors can stimulate the cones to respond in a way that Is indistinguishable from a pure color.
- Perceptually identical sensations are called metamers.
- This allows us to use just three colors to generate all others.





How colors Are Represented

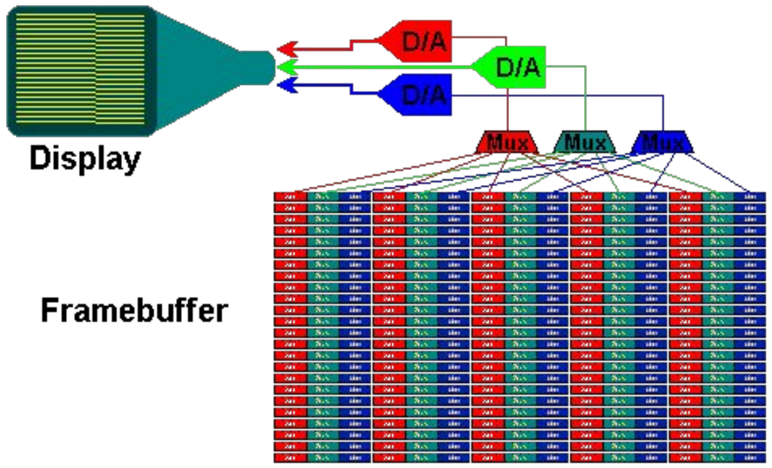
- Each pixel is stored as three primary parts
- Red, green, and blue
- Usually around 8-bits per channel
- Pixels can have individual R,G,B components or they can be stored indirectly via a "look-up table"

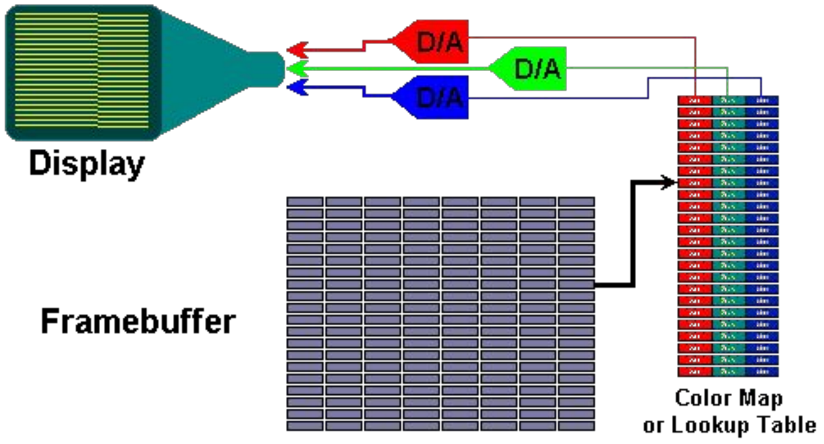
8-bits	8-bits	8-bits

3 - 8-bit unsigned binary integers (0,255)

-OR-

3 - fixed point 8-bit values (0-1.0)





Color Specifications

Web colors:

Name	Hex	Decimal Integer	Fractional
Orange	#FFA500	(255, 165, 0)	(1.0, 0.65, 0.0)
Sky Blue	#87CEEB	(135, 206, 235)	(0.52, 0.80, 0.92)
Thistle	#D8BFD8	(216, 191, 216)	(0.84, 0.75, 0.84)

Colors are stored as binary too. You'll commonly see them in Hex, decimal, and fractional formats.

Summary

- **ALL** modern computers represent signed integers using a two's-complement representation
- Two's-complement representations eliminate the need for separate addition and subtraction units
- Addition is **identical** using either unsigned and two's-complement numbers
- Finite representations of numbers on computers leads to anomalies
- Floating point numbers have separate fraction and exponent components.