BINARY NUMBERS AND OTHER OBJECTS

Eric Prebys

“Numbers” in Computing

- As you’ll learn if you take 118, everything in a computer ultimately reduces to 1s (TRUE) and 0s (FALSE)
  - These are internally defined by voltage levels.
- Numbers are made from groups individual signals, called “bits” (= “binary digit”)
- In the early days of computing, the exact format of numbers depended on the type of computer and there was a lot of variation:
  - Even “byte” was only defined as ”a group bits” when I was in college (sometimes 6, sometimes 8)

Types of Numbers

- In general, there are four types of numbers:
  - Unsigned Integers
  - Signed Integers
  - Fixed Point (really a special case of Signed Integers)
  - Floating Point
- Unsigned Integers are the only one with an unambiguous definition.
- Signed Integers and floating-point numbers have used different representations over the years.
- We'll focus on modern standards, but it's not impossible that you may encounter older standards somewhere down the road.

Unsigned integers (also used for addressing!)

- The binary representation of unsigned integers is simply the number expressed in base-2, in which the n^{th} bit represents 2^n (we'll always count bits starting at 0!)

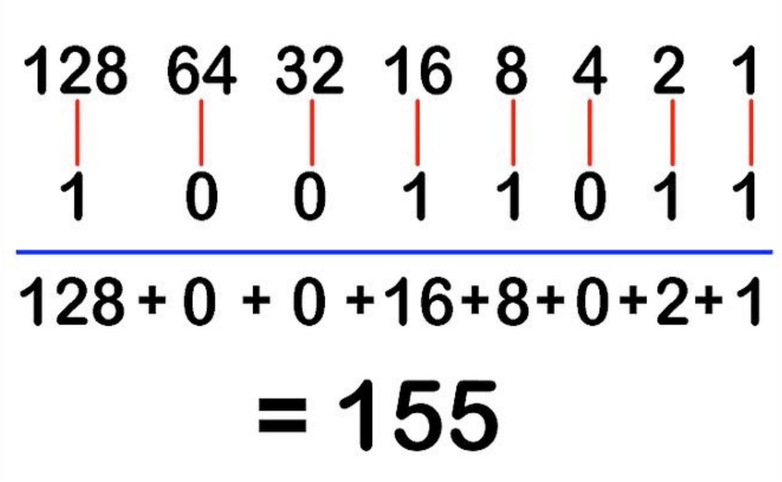
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7

1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

- A *n*-bit word has a range 0 to (2^n-1)

Converting from Binary to Decimal

- Simply add up the values of the individual digits



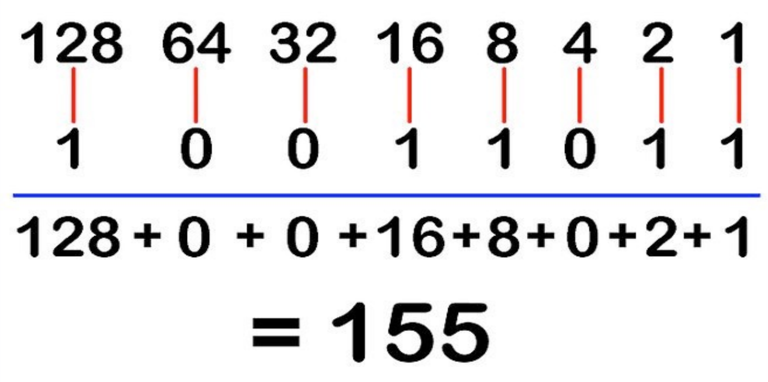
128	64	32	16	8	4	2	1
1	0	0	1	1	0	1	1

128 + 0 + 0 + 16 + 8 + 0 + 2 + 1

= 155

Converting from Decimal to Binary

- Recursive subtraction
  - Find the largest value of *n* for which 2^n is less than the number
  - Set bit *n*
  - Subtract 2^n from original number
  - Repeat until the remainder is zero or $n=0$ (ie, remainder=1)
- Using our previous example (155)
  - $2^7=128$ is the largest power of 2 smaller than 155. Set bit 7. 155-128=27
  - $2^4=16$ is the largest power of 2 smaller than 27. Set bit 4. 27-16=11
  - $2^3=8$. Set bit 3. 11-8=3
  - $2^1=2$. Set bit 1. 3-2=1
  - $2^0=1$. Set bit 0



- Or just ask Google!

Signed Integers

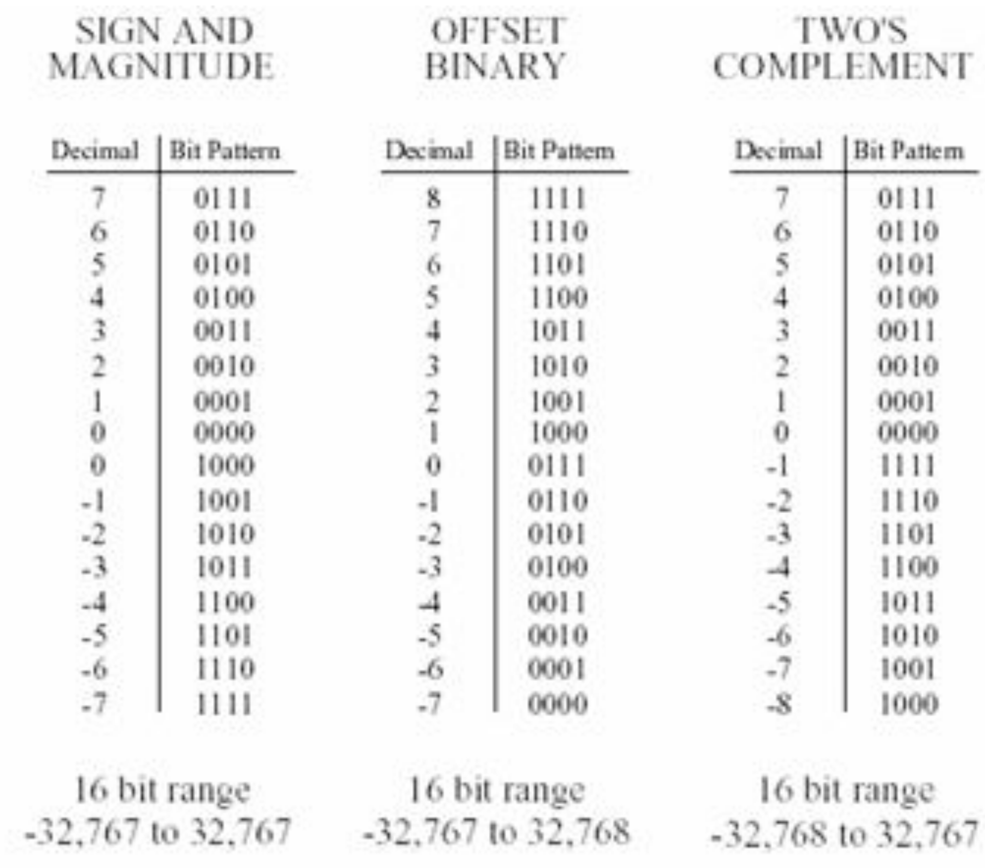
- The three most common ways of representing signed integers
  - Sign and magnitude
    - Very old computers
    - The mantissa of floating point numbers
  - Offset binary

    $$N' = N + \left(2^{n-1} - 1\right)$$

    - Used for some hardware
    - Used for exponent in floating point
  - Twos Complement

    $$-N = 2^n - N$$

    - Standard in modern computers
    - Addition is always the same



SIGN AND MAGNITUDE		OFFSET BINARY		TWO'S COMPLEMENT	
Decimal	Bit Pattern	Decimal	Bit Pattern	Decimal	Bit Pattern
7	0111	8	1111	7	0111
6	0110	7	1110	6	0110
5	0101	6	1101	5	0101
4	0100	5	1100	4	0100
3	0011	4	1011	3	0011
2	0010	3	1010	2	0010
1	0001	2	1001	1	0001
0	0000	1	1000	0	0000
0	1000	0	0111	-1	1111
-1	1001	-1	0110	-2	1110
-2	1010	-2	0101	-3	1101
-3	1011	-3	0100	-4	1100
-4	1100	-4	0011	-5	1011
-5	1101	-5	0010	-6	1010
-6	1110	-6	0001	-7	1001
-7	1111	-7	0000	-8	1000
16 bit range -32,767 to 32,767		16 bit range -32,767 to 32,768		16 bit range -32,768 to 32,767	

Binary addition

- Binary addition is just like decimal addition except that you carry a whole lot more. 8-bit example

$$\begin{array}{rr} & 00\overset{\curvearrowleft}{1}00\overset{\curvearrowleft}{0}\overset{\curvearrowleft}{1}\overset{\curvearrowleft}{1} \\ + & 10110111 \\ \hline & 11011010 \end{array}$$

- Addition can ”overflow”, in which case, higher order bit(s) will be ignored

$$\begin{array}{rr} & 10100011 \\ + & 10110111 \\ \hline & \cancel{1}01011010 \end{array}$$

This would be dropped for 8-bit words

Two’s Complement

- An *n*-bit negative number is formed by subtracting the number from 2^n
- In additions, overflow bits are ignored
- Example: -55 (-110111_2) in 8 bits

$$2^8 - 55 = 256 - 55 = 201 = 11001001_2$$

- Trick: just invert all bits (“1’s complement”) and add 1!
- Check

$$\begin{array}{rr} & 00110111_2 \\ + & 11001001_2 \\ \hline & \cancel{1}00000000_2 \end{array}$$

To extend to higher number of bits, repeat MSB
In 16 bits

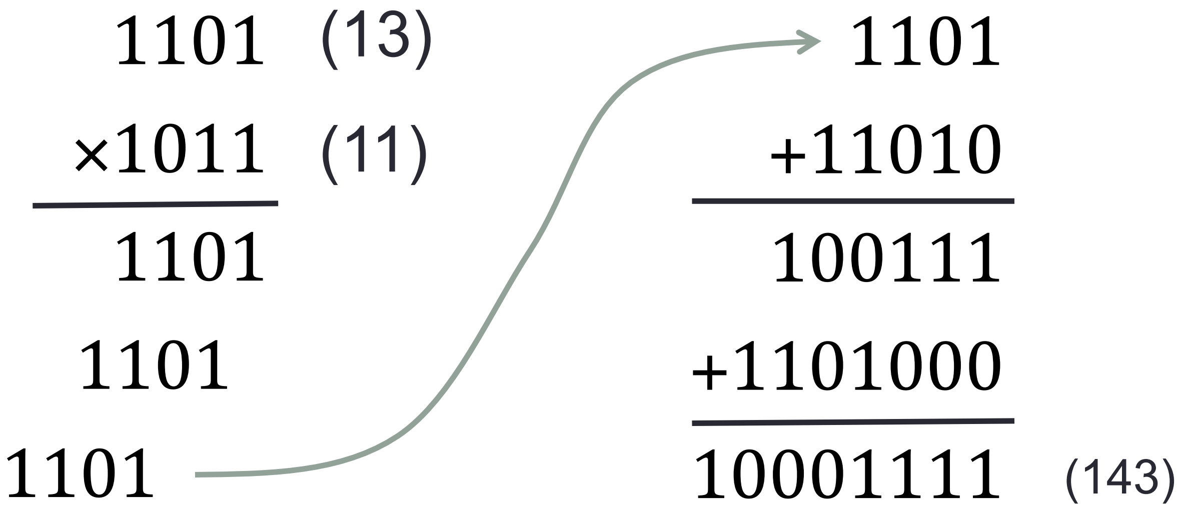
$$55 = 0000000000110111_2$$

$$-55 = 1111111111001001_2$$

Binary Multiplication

- Binary multiplication is just like decimal multiplication, except that multiplying something by *2ⁿ* just shifts it *n* bits to the left.

$$1011011_2 \cdot 100_2 = 101101100_2$$

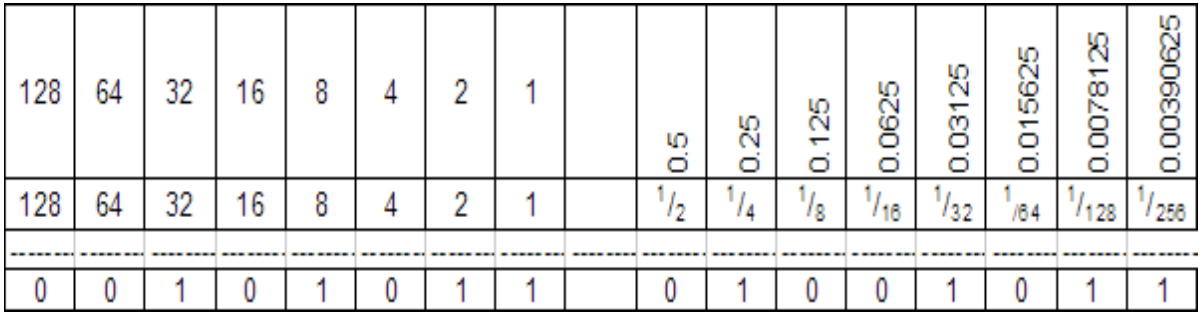


$$\begin{array}{rl} 1101 & (13) \\ \times 1011 & (11) \\ \hline 1101 & \\ 1101 & \\ 1101 & \end{array} \qquad \begin{array}{rl} 1101 & \\ +11010 & \\ \hline 100111 & \\ +1101000 & \\ \hline 10001111 & (143) \end{array}$$

- Note! Just like decimal multiplication, multiplying two n-bit numbers can result in a 2n-bit number -> easy to overflow

Fixed Point Numbers

- Fixed point numbers are just an offset case of signed integers

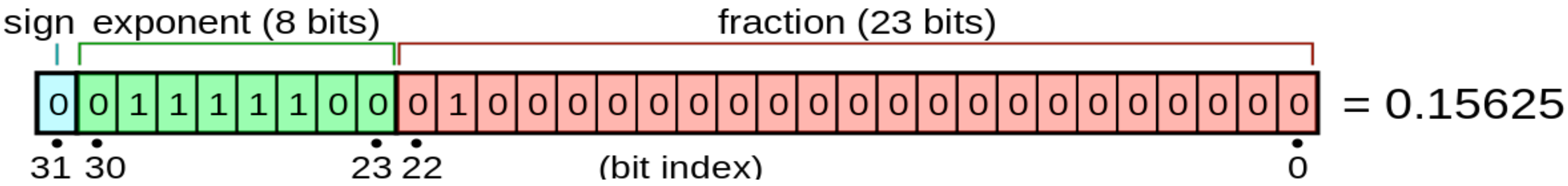


128	64	32	16	8	4	2	1		0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	0.00390625
128	64	32	16	8	4	2	1		1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256
0	0	1	0	1	0	1	1		0	1	0	0	1	0	1	1

- Example: 37.55
  - Get roughly equivalent precision by expressing in 128ths (7 bits), but it's common to work in groups of 4 (16ths) or 8 (256ths), so let's use 8 bits after the radix:
    - Multiply by 2^8 (256) $37.55 \times 256 = 9613$
    - Convert to binary $9613_{10} = 10010110001101_2$
    - Shift radix left by 8 $\underbrace{100101}_{37}.\underbrace{10001101}_{\frac{141}{256} = .551}$

Floating Point Numbers (IEEE 754*)

- Single precision (32-bit)



sign exponent (8 bits) fraction (23 bits)

0 0 1 1 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 = 0.15625

31 30 23 22 (bit index) 0

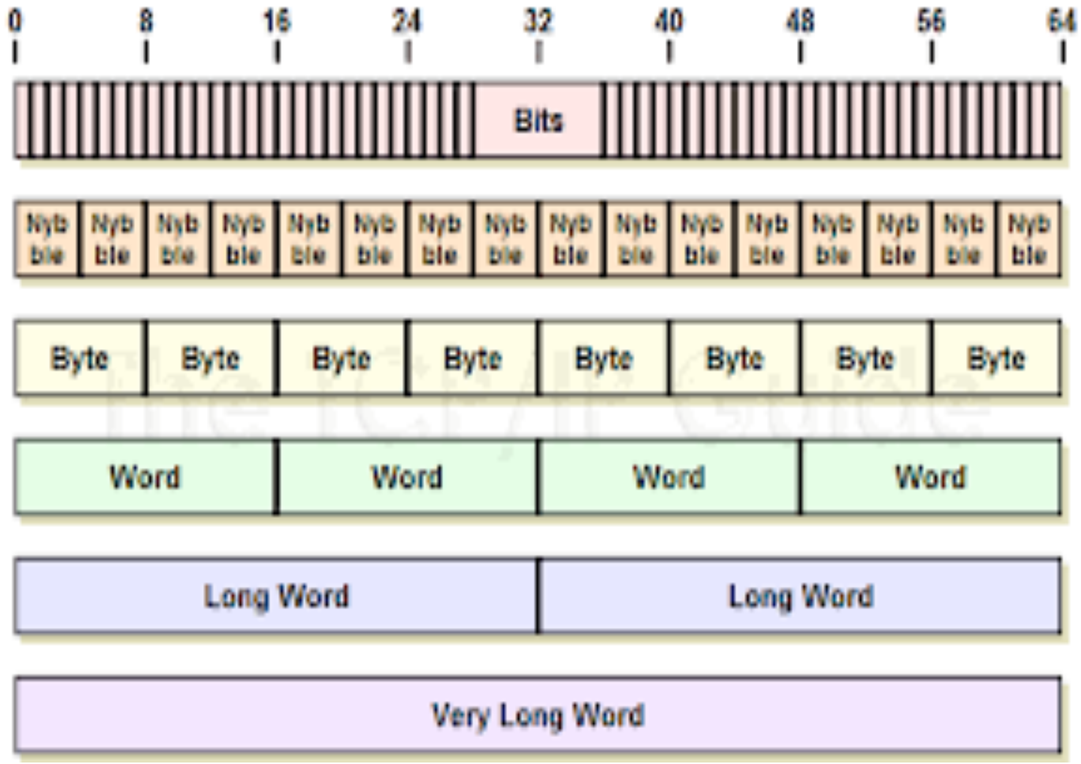
$$(-1)^{b_{31}} \times (1.b_{22}b_{21}\ldots b_0)_2 \times 2^{(b_{30}b_{29}\ldots b_{23})_2-127}$$

Leading 1 assumed (why?)

- IEEE 754 includes standards from 16-bit (half precision) to 256-bit (octuple precision) floating point.
- Special values
  - exponent bits set to zero -> zero (also allows for -0!)
  - Exponent bits all set and all fraction bits clear -> Infinity
  - Exponent bits all set and *some* fraction bits set -> NaN

*became a standard in 1985

Grouping Bits in Modern Computers



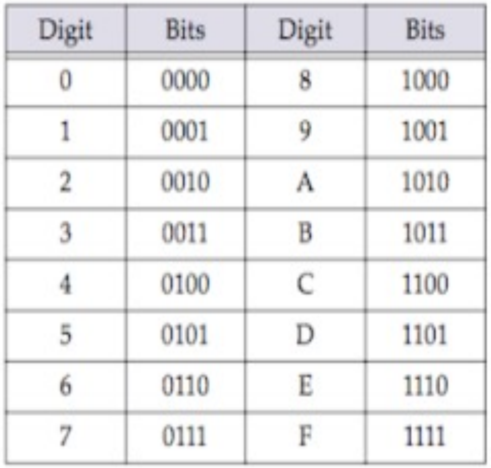
This took a long time to evolve. Examples

- Some old computers (IBM, PDP) had 6-bit bytes
- CDC mainframes had 60-bit words, with totally different encoding.

- Hardware will generally not group bits exactly like this
  - Still tend to group things in 8-bit bytes
  - The term “word” is used generically

Representing Groups of Bits

- Regardless of their use, groups of bits can always be compactly represented with hexadecimal numbers
  - Each byte is represented by a two-digit hex number 00->FF, where each digit represents 4 bits

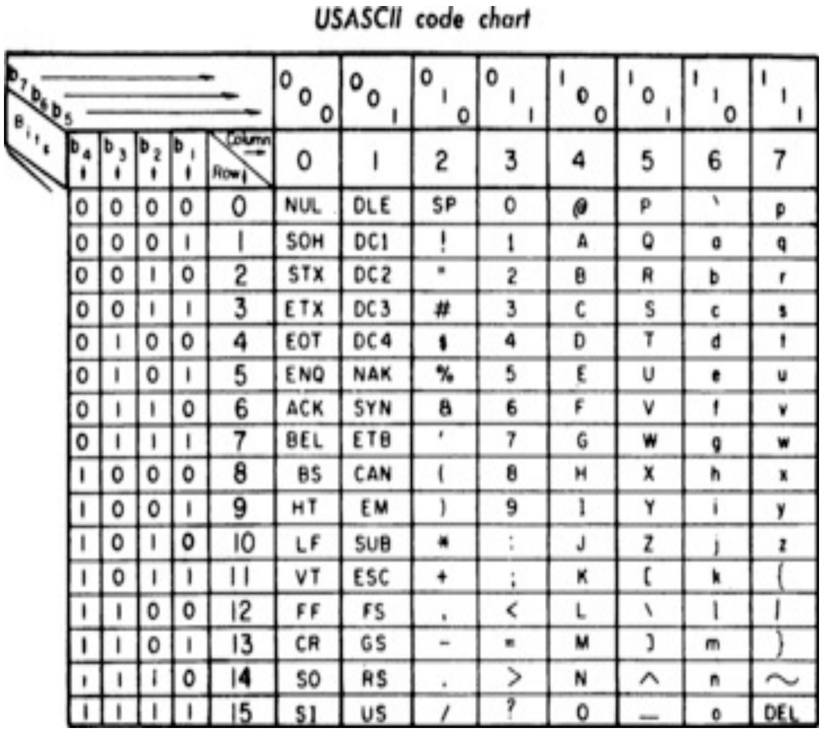


Digit	Bits	Digit	Bits
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

- We can then represent larger groups of bits by larger hex numbers, while still respecting byte boundaries
  - One byte is two hex digits
  - Historically, 6-bit bytes were represented by two digit octal.

Representing Characters

- The most common character encoding is American Standard Code for Information Interchange (ASCII), in which each 8-bit byte represents one character



USASCII code chart

Bits				b_7 b_6 b_5 →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 ↓	b_3 ↓	b_2 ↓	b_1 ↓	Column → Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	–	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

- In recent years, the 16-bit "Unicode" standard has emerged to accommodate foreign alphabets as well as a wide range of special characters.

Evolution of Precision

- Numbers used to default to 16-bit precision
  - Unsigned range: 0 to 65,535
  - Signed range: -32,768 to +32,767
  - This was standard for PC until the mid-1990s
- All systems eventually moved to 32-bit
  - Unsigned range: 0 to ~4 billion
  - Signed range: -2 billion to +2 billion
- All modern systems are 64-bit
  - Unsigned range: 0 to 18 quintillion
  - Signed range: -9 quintillion to 9 quintillion
- The current release of Python is 64-bit, but Python handles integers in a special way that allows arbitrarily large range.

Integer Mathematical Precision

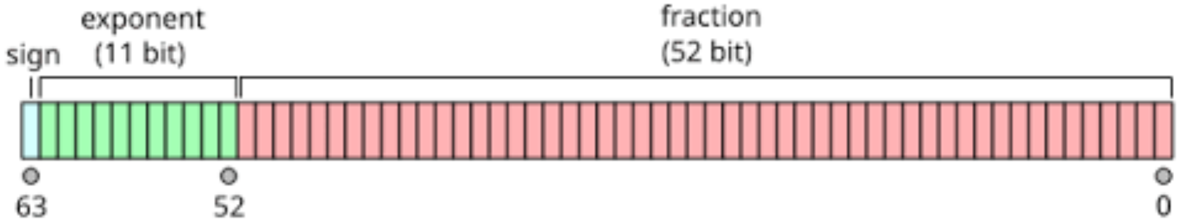
- Integer math is exact!
  - You’ll always get the correct answer until it overflows.
    - For signed 64-bit this is a pretty large number, but we deal with very large numbers.
    - Can happen pretty quickly with multiplication.
- Python “int” objects have arbitrary size.
  - Necessary size is determined when the object is initialized
  - Numerous internal methods give details about the structure
  - Integer operations will *never* overflow or have round-off errors!
  - (Jupyter example shortly)

Important differences between Python and Other OO Languages…

- Both C++ and Java distinguish between “primitive” data types and “objects”
  - Primitives: integer, floating point, character, etc. These have no inherent methods!
  - Objects: More complex structures that include data and associated methods.
- Python has *only* objects, even for simple data types:
  - Everything has methods.
  - This leads to more versatility
  - The distinction is between “mutable” and “immutable”, which we’ll discuss shortly.

Floating Point Precision in Python

- While C/C++ and Java distinguish between ”float” (32-bit) and “double” (64-bit) numbers, 64-bit Python “float” is *always* 64-bit.



- Range: $\pm 4.9 \times 10^{-324}$ to $\pm 1.8 \times 10^{308}$
- Precision: 52 binary digits ~15 decimal digits
- Multiplication
  - Add exponents
  - Multiply fractions (including implicit 1)
  - Still only have 52-bit precision
- Addition
  - Right-shift smaller number so exponent matches larger number
  - Add fractions (including implicit 1)
  - Will round off if one number is > ~2*53 smaller than the other.

Other Number Types

- When it comes to built-in number types, Python has ONLY
  - **int**: unlimited size integers (math *extremely* inefficient)
  - **float**: floating point with precision of the OS (now always 64-bit)
- Sometimes, we'll want other things
  - Integers of fixed bit length
  - Floating point numbers with different bit lengths (32 or 128)
  - This is *extremely* important when calling routines in C/C++ and other languages.
- This functionality will be provided by the "Numerical Python" (numpy) library, which we'll introduce shortly.
- Boolean (bool) variable
  - Values of "True" or "False". Can be thought of as a 1-bit integer

tuples, lists, and keyed lists (dicts)

- Ordered sets:
  - ”tuples” are ordered sets of objects in which individual items CANNOT be modified
  - “lists” are ordered sets of objects in which individual items.
    - Lists of numbers are referred to as “arrays”, but we’ll do our array math with numerical python (numpy)
- Keyed sets
  - Sets in which elements are addressed by keyword are called “keyed lists” or “dicts”

Mutable vs. Immutable

- Simple objects in Python are “immutable”, meaning they can’t be changed once created.
  - This is generally associated with “pass by value”, although the way Python implements it is kind of weird.
- More complex objects are “mutable”, meaning that objects can be changed once they are created
  - When on object is equated to another, it is “passed by reference” rather than creating a new object.

(go to notebook demos)