CSE 374 Programming concepts and tools

Winter 2024 Instructor: Alex McKinney

Review: Numbers Can Represent Anything

Text files

- "ASCII": uses one byte to represent a single character; Each number corresponds to a different character
- "Unicode": similar encoding structure to ASCII but covers a wider range of characters including non-English characters, emojis etc…
  - 好, あ, 😀 (2+ bytes to represent)

Images: represented by a 2D array of “pixels”

- Each pixel is represented by 3 numbers: Red, Blue and Green values 0-255

Data

Characters

ASCII (see asciitable.com)

ASCII (American Standard Code for Information Interchange) is a character encoding standard used to represent text in computers and communication devices.

- Uses one byte to represent a single character

Each ASCII character is represented by a unique numerical value, ranging from 0 to 127.

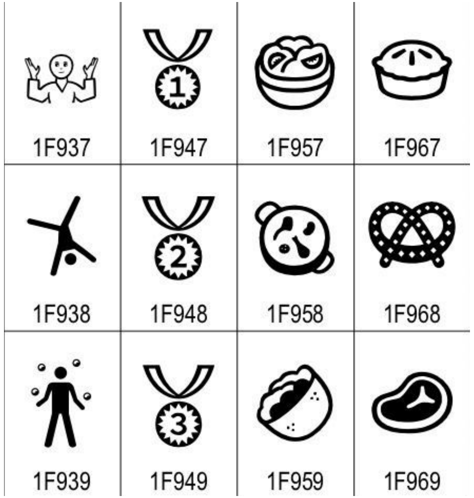
Each number corresponds to a different character, such as uppercase and lowercase letters, digits, and common symbols.

- e.g. 65 = '`A`', 66 = '`B`', …
- e.g. 32 = '` `', 33 = '`!`'

Character Encoding

The ASCII table maps each character to its corresponding numerical value, allowing computers to interpret and display text.

- Works well for languages using a latin-based alphabet



In practice, any modern application should expect Unicode (UTF8)

In CSE 374, we will only work with ASCII, since it is simpler

Special ASCII Characters

Some numbers in ASCII do **not** correspond to a character

Instead, they have a special meaning

- 4 = End of Transmission (hitting Control+D to close `stdin`)
- 10 = New line (written as '`\n`')
- 0 = Null (written as '`\0`')

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Converting between `char` and `int` in C

Converting from `char` to `int`

- When a char is assigned to an int variable, the ASCII value of the character is implicitly converted to its corresponding integer value.
- Example: `char myChar = 'A'; int myInt = myChar;`
  - Assigns the ASCII value of 'A' (65) to myInt.

Converting from `int` to `char`

- When an int is assigned to a char variable, the integer value is implicitly converted to its corresponding ASCII character.
- Example: `int myInt = 65; char myChar = myInt;`
  - Assigns the ASCII character 'A' to myChar.
  - It's possible to overflow if myInt > 255. Usually a good idea check the bounds

Demo: Converting b/w char and int

Questions?

Bonus Challenge

This is an example of a real interview question given by a Google engineer!

Question: Given a string that is composed of every letter of the English alphabet except one, how do you efficiently find the missing character?

Bonus Challenge

This is an example of a real interview question given by a Google engineer!

Question: Given a string that is composed of every letter of the English alphabet except one, how do you efficiently find the missing character?

Follow-up: What if you're not allowed to use a dictionary or a hashmap?

Bonus Challenge

This is an example of a real interview question given by a Google engineer!

Question: Given a string that is composed of every letter of the English alphabet except one, how do you efficiently find the missing character?

Follow-up: What if you're not allowed to use a dictionary or a hashmap?

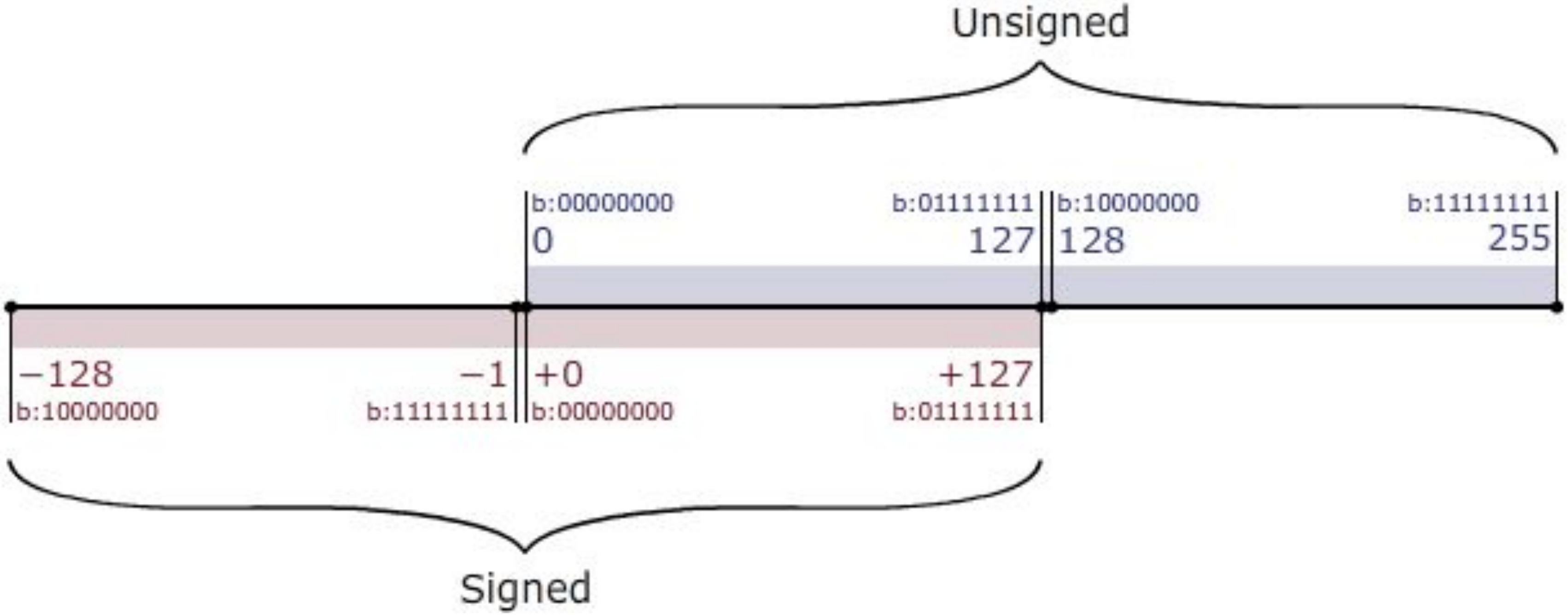
Answer: Sum the ASCII values contained in the string, then subtract it from the total value of the ASCII alphabet.

```
sum(abcdefghijklmnopqrstuvwxyz) == 2847

sum(abdefghijklmnopqrstuvwxyz) == 2748

2847 - 2748 == 99 == 'c'
```

Example: 8-bit integers



Integers

Review: Number systems and BASE

Generally use base 10

(decimal)

374

$3x100 + 7x10 + 4x1$

$3x10^2 + 7x10^1 + 4x10^0$

Digital systems - base 2

(binary)

374 = **0b**101110110

$1x2^8 + 0x2^7 + 1x2^6 + 1x2^5 +$
$1x2^4 + 0x2^3 +1x2^2+ 1x2^1 +$
$0x2^0$

Base 16 - very compact

(hexadecimal)

374 = **0x**176

$1x16^2 + 7x16^1 + 6x16^0$

Need 16 digits,
so we used [0-9A-F]

Notice: 374 takes 3 digits to express in base 10, 9 in base 2, and 3 in base 16.

Integer representations

The hardware (and C) supports two flavors of integers

- unsigned – only the non-negatives, follow standard base 2 system
  - The number systems we've seen so far have been unsigned
- signed – both negatives and non-negatives
  - Signed numbers are stored slightly differently

There are only 2^W distinct bit patterns of W bits, so we cannot represent all the integers

- Unsigned values: **$0 \ldots 2^W-1$**
  - Example (4 bits): 2^4-1 -> 1111 -> $2^3+2^2+2^1+2^0$ -> 8+4+2+1 -> 15
- Signed values: **$-2^{W-1} \ldots 2^{W-1}-1$**

In the C language, `int` means signed (positive or negative)

- You can force unsigned (e.g. `unsigned int x = 42;`)

Terminology for Binary Representations

The Most Significant Bit (**MSB**) is the leftmost bit in a binary representation, while the Least Significant Bit (**LSB**) is the rightmost bit.

MSB LSB

0b01110110

How to represent signed integer?

Not used in practice for integers

Obvious solution: designate the MSB as the "sign bit"

- `sign=0`: positive numbers; `sign=1`: negative numbers

Benefits:

- Using MSB as sign bit matches positive numbers with unsigned
- All zeros encoding is still = 0

Examples (8 bits):

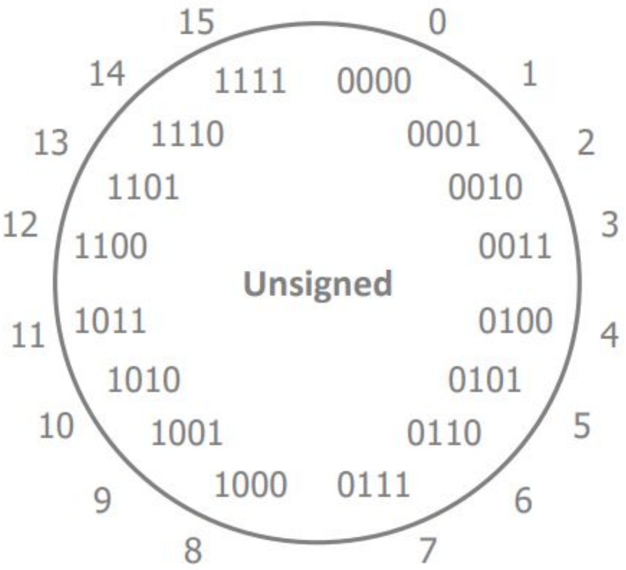
- 0x00 = 0b00000000 is non-negative, because the sign bit is 0
- 0x7F = 0b01111111 is non-negative (+127)
- 0x85 = 0b10000101 is negative (-5)
- 0x80 = 0b10000000 is negative ... 0?

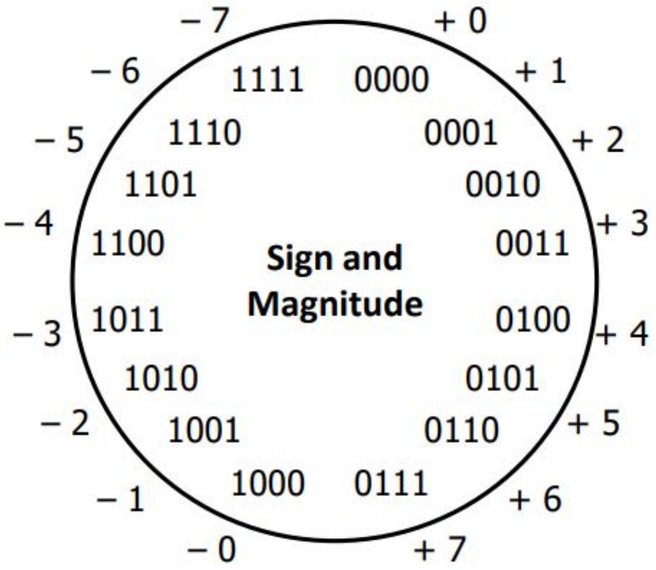
Sign and Magnitude

Not used in practice for integers

MSB is the sign bit, rest of the bits are magnitude

Drawbacks?



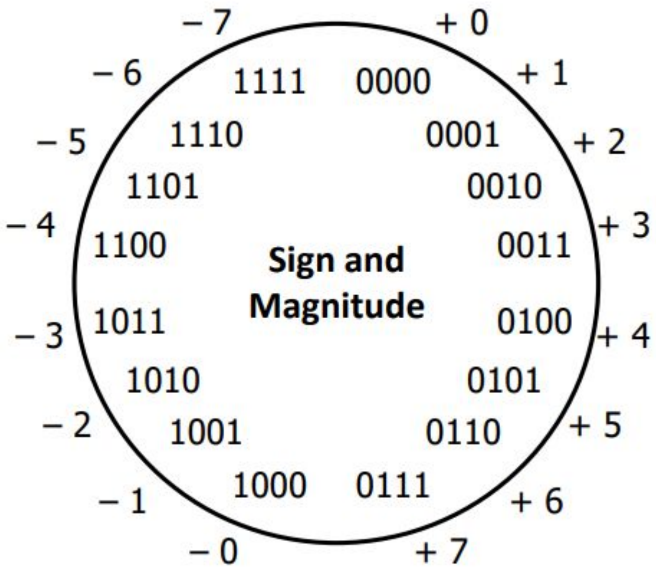


Sign and Magnitude

Not used in practice for integers

Drawbacks?

- Two representations of **0** (bad for checking equality)
- **Arithmetic** is cumbersome
  - Negatives “increment” in wrong direction!
  - Example: 4 - 3 != 4 + (-3)



Adding unsigned ints
(add and carry normally)

```
 0101
+0011
------
 1000
```

Adding signed ints
(gets tricky)

```
 0100   (4)
+1011   (-3)
------
 1111     = -7 ?
```

One's Complement

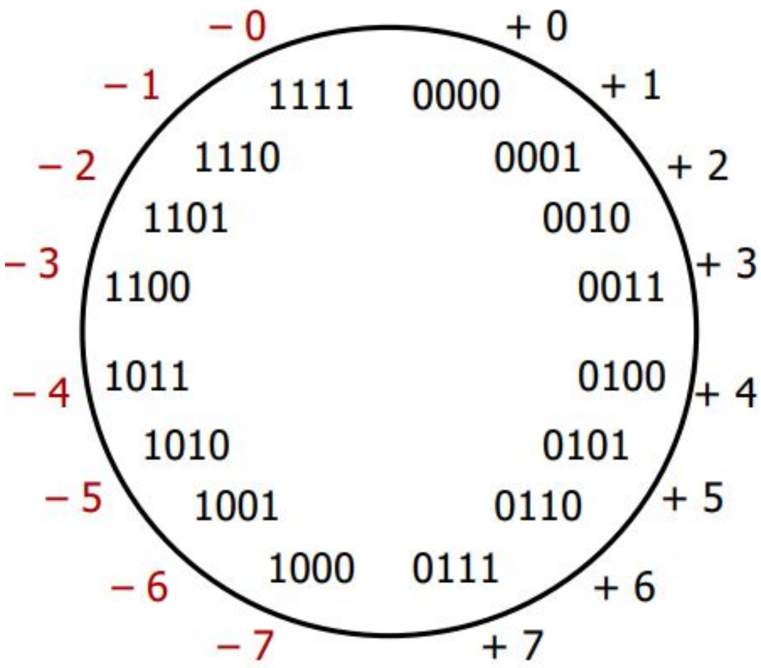
Let's fix these problems:

1. "Flip" negative encodings so incrementing works

0111 == 7

1000 = -7

Arithmetic works again, but we still have two 0's…



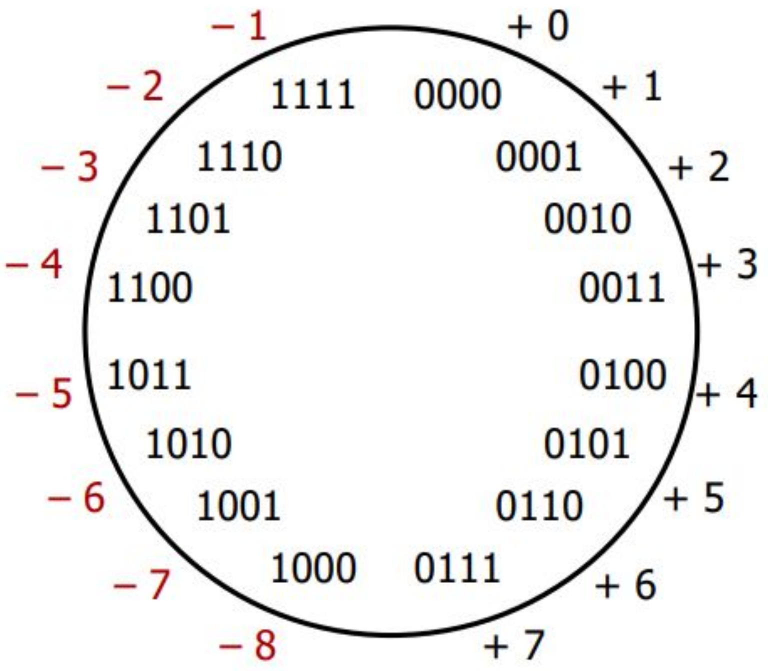
Two's Complement

Let's fix these problems:

1. "Flip" negative encodings so incrementing works
2. "Shift" negative numbers to eliminate -0

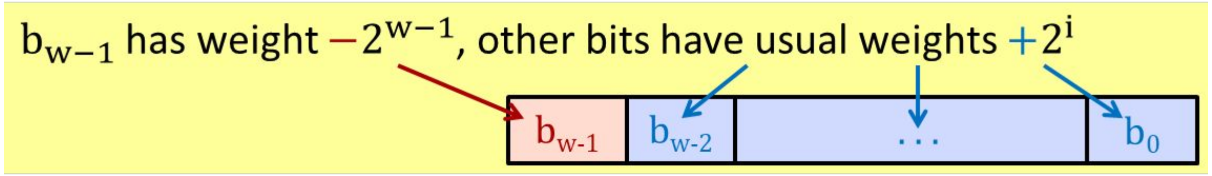
MSB **still** indicates sign!

- This is why we represent one more negative than positive number ($-2^{W-1} \ldots 2^{W-1}-1$)

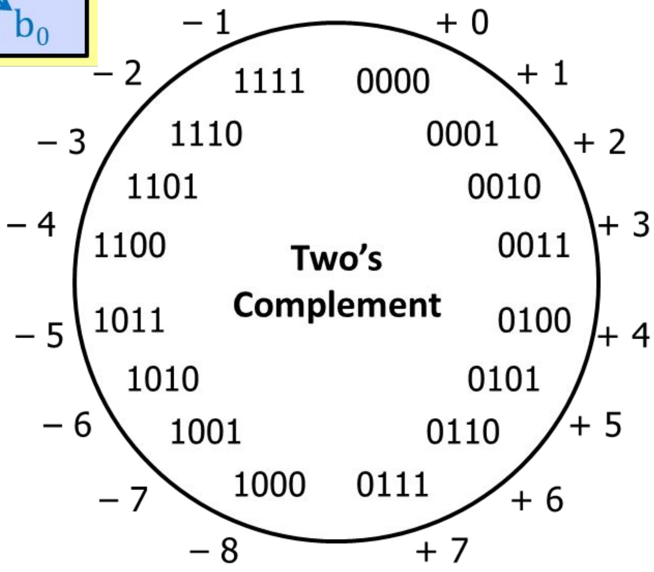


Two's Complement Negatives

Accomplished with one neat mathematical trick!

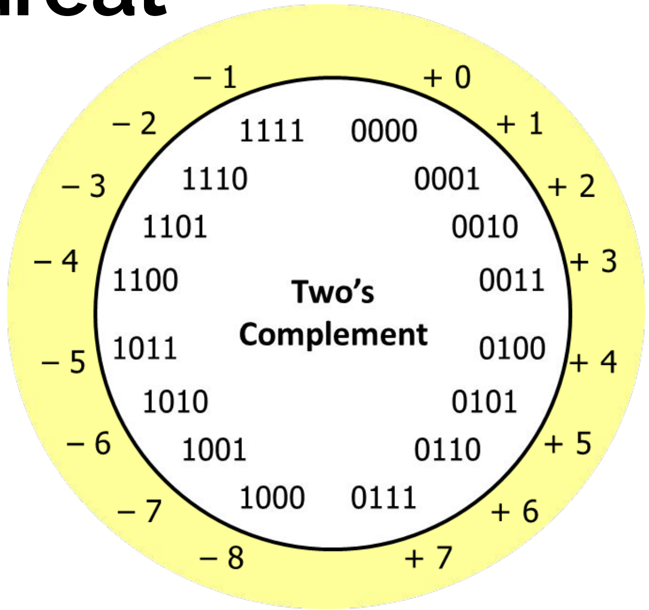


- 4-bit Examples:
  - 1010_2 unsigned: $1*2^3+0*2^2+1*2^1+0*2^0$ = **10**
  - 1010_2 two's complement: $-1*2^3+0*2^2+1*2^1+0*2^0$ = **–6**
- -1 represented as: $1111_2 = -2^3+(2^3 - 1)$
- MSB makes it super negative, add up all the other bits to get back up to -1



Why Two's Complement is So Great

- Only 1 representation of 0
- MSB is still the sign
- Simple negation procedure:
- Take bitwise complement and then adding one!
    - `~x + 1 == -x`
- Roughly same number of (+) and (–) numbers
- Positive number encodings match unsigned
- Adding becomes easy again
    - Example: `0100 + 1101 = 0001`
        - 4 - 3 = 4 + -3 = 1




```
 0100
+1101
------
 0001   = 1
```

Poll Question pollev.com/cs374

Take the 4-bit number encoding `x = 0b1011`

Which of the following numbers is NOT a valid interpretation of x using any of the number representation schemes discussed today?

- Unsigned, Sign and Magnitude, Two's Complement

A) -4
B) -3
C) -5
D) 11

Consider x = 0b1011; which of the following is NOT valid?

Poll Question (Explained)

Take the 4-bit number encoding `x = 0b1011`

Which of the following numbers is NOT a valid interpretation of x using any of the number representation schemes discussed today?

- Unsigned, Sign and Magnitude, Two's Complement

Unsigned: 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 8 + 2 + 1 = **11**

Sign and magnitude: -(0*2^2 + 1*2^1 + 1) = -(2 + 1) = **-3**

Two's complement: 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = -8 + 2 + 1 = **-5**

What happens if you 'overflow'

Overflow: have numbers too big or small for your number of digits.

Remember, using 4 bits, unsigned = [0,15] and signed [-8,7]

6+4 = ? (signed) 15+2 = ? (unsigned)

6 - 8 = ? (signed) 12-14 = ? (unsigned)

Notes: You may get a warning for overflow with two-complement numbers, but probably not with unsigned numbers.

```
 0110          1111
+0100         +0010
------        ------
 1010 (-6!)    0001 (1!)
```

```
 0110          1100
-1000         -1110
------        ------
 1110 (-6!)    1110 (14!)
```

In C: Signed vs. Unsigned

Casting: bits are unchanged, just interpreted differently! This is NOT taking the absolute value.

- `int tx, ty;`
- `unsigned ux, uy;`

Explicit casting between signed & unsigned:

- `tx = (int) ux;`
- `uy = (unsigned) ty;`

Implicit casting also occurs via assignments and function calls:

- `tx = ux;`
- `uy = ty;`

Questions?

Floating Point Numbers

Real numbers (e.g. 3.14159)

Very large numbers (e.g. 6.02×10^{23})

Very small numbers (e.g. 6.626×10^{34})

Special numbers (e.g. ∞, NaN)

Floating Point Numbers

All the numbers we have seen so far have been integers

- i.e. No decimal point: 42 vs 1.5
- Remember `int` vs `double` from Java?

To store numbers with a decimal point, we need a different way of storing numbers

Floats are stored kind of like scientific notation numbers

- e.g. $0.123 = 1.23 * 10^{-1}$

Scientific Notation (Decimal)

exponent

$$\overbrace{6.02}^{\text{mantissa}}{}_{10} \times 10^{23}$$

decimal point

radix (base)

- *Normalized form*: exactly one digit (non-zero) to left of decimal point
- Alternatives to representing 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: 0.1×10^{-8}, 10.0×10^{-10}

Scientific Notation (Binary)

exponent

$$\overbrace{1.01}^{\text{mantissa}}{}_2 \times 2^{-1}$$

binary point

radix (base)

- Computer arithmetic that supports this called floating point due to the “floating” of the binary point
  - Declare such variable in C as `float` (or `double`)

Scientific Notation Translation

Floating point values only represent numbers that can be written $x \cdot 2^y$

Convert from scientific notation to binary point

- Perform the multiplication by shifting the decimal until the exponent disappears
  - Example: $1.011_2 \times 2^4 = 10110_2 = 22_{10}$
  - Example: $1.011_2 \times 2^{-2} = 0.01011_2 = (1/4 + 1/16 + 1/32)_{10} = 0.34375_{10}$

Convert from binary point to *normalized* scientific notation

- Distribute out exponents until binary point is to the right of a single digit
  - Example: $1101.001_2 = 1.101001_2 \times 2^3$

IEEE Floating Point

IEEE 754

- Established in 1985 as uniform standard for floating point arithmetic
- Main idea: make numerically sensitive programs portable
- Now supported by all major CPUs

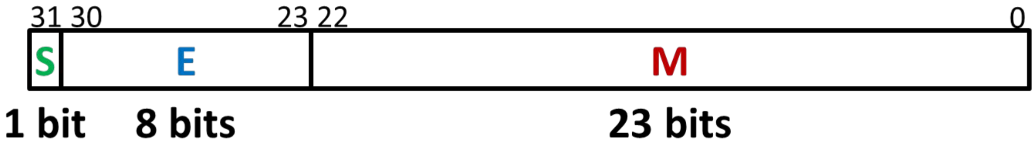
Driven by numerical concerns

- **Scientists**/numerical analysts want them to be as **real** as possible
- **Engineers** want them to be **easy to implement** and **fast**
- In the end: Scientists mostly won out
- Nice standards for rounding, overflow, underflow, but…
  - Hard to make fast in hardware
  - Float operations can be an order of magnitude slower than integer ops

Floating Point Encoding

Use normalized, base 2 scientific notation:

- Value: $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$
- Bit Fields: $(-1)^{S} \times 1.M \times 2^{(E-\text{bias})}$



Representation Scheme:

- Sign bit (0 is positive, 1 is negative)
- Mantissa (a.k.a. significand) is the fractional part of the number in normalized form and encoded in bit vector **M** , [1.0, 2.0)
- Exponent weights the value by a (possibly negative) power of 2 and encoded in the bit vector **E**

The Exponent Field

Use biased notation

- Read exponent as unsigned, but with *bias* of $2^{w-1}-1$ = 127
- Representable exponents roughly ½ positive and ½ negative
- Exponent 0 (Exp = 0) is represented as E = 0b 0111 1111

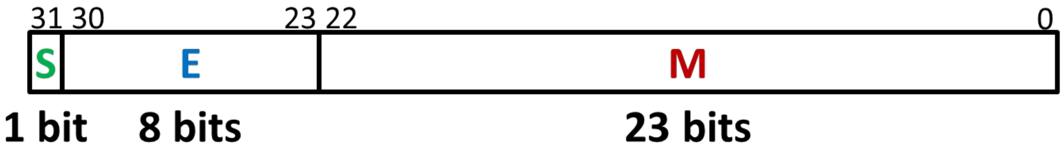
Why biased?

- Makes floating point arithmetic easier
- Simplified ordering and comparison
- Makes somewhat compatible with two's complement

Practice: To encode in biased notation, add the bias then encode in unsigned:

- Exp = 1 $\rightarrow$ 128 $\rightarrow$ E = 0b 1000 0000
- Exp = -63 $\rightarrow$ 64 $\rightarrow$ E = 0b 0100 0000

The Mantissa (Fraction) Field



$$(-1)^{S} \text{ x } (1\ .\ M) \text{ x } 2^{(E-\text{bias})}$$

Note the **implicit 1** in front of the M bit vector

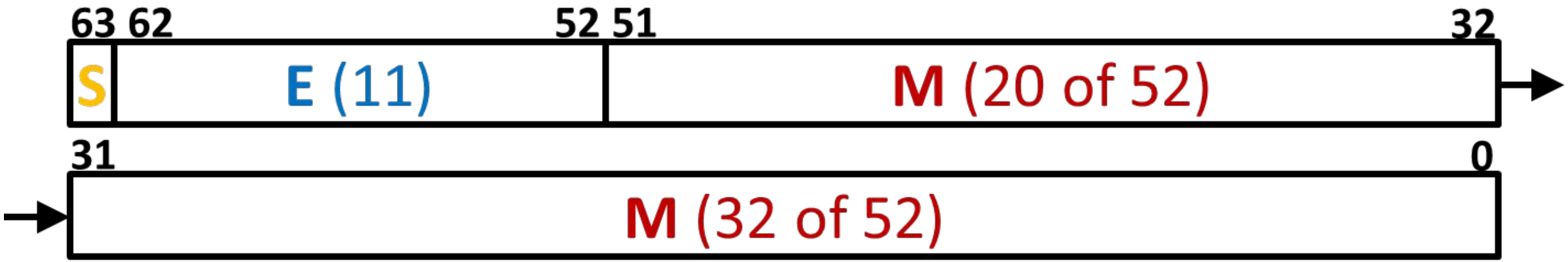
- Example: 0b 0011 1111 1100 0000 0000 0000 0000 0000 is read as $1.1_2 = 1.5_{10}$, *not* $0.1_2 = 0.5_{10}$
- Gives us an extra bit of ***precision***

Special Values, which can pollute numerical computations

- zero: S == 0, E == 0, M == 0
- +∞, -∞: E == all ones, M == 0
- NaN (not a number): E = all ones, M != 0

Need Greater Precision?

Double Precision (vs. Single Precision) in 64 bits



- C variable declared as `double`
- Exponent bias is now $2^{10}-1 = 1023$
- **Advantages:** greater precision (larger mantissa), greater range (larger exponent)
- **Disadvantages:** more bits used, slower to manipulate

Floating Point in C

- Two common levels of precision:

`float`	`1.0f`	single precision (32-bit)
`double`	`1.0`	double precision (64-bit)

- `#include <math.h>` to get `INFINITY` and `NAN` constants
- Equality (==) comparisons between floating point numbers are tricky
  - Often return unexpected results
  - Just avoid them!

Floating Point Summary

Floats also suffer from the fixed number of bits available to represent them

- Can get overflow/underflow, just like ints
- Can also lose precision, unlike ints.
  - Some “simple fractions” have no exact representation (e.g., 0.2)
  - “Every operation gets a slightly wrong result”

Floating point arithmetic not associative or distributive

- Mathematically equivalent ways of writing an expression may compute different results

Never test floating point values for equality!

Careful when converting between `int`s and `float`s!

Aside: Don’t use `float` for money!

Given the approximate nature of `float`, it’s best not to use it for *precise* measurements.

Banks actually use integers to represent money (i.e. in *cents*).

$1.01 == 101 units

In general, prefer `int` over `float` whenever you need absolute precision!

Questions?

Data type conversions

Casting between `int`, `float`, and `double` changes the bit representation.

- `int` → `float`
  - May be rounded (not enough bits in mantissa)
  - Overflow impossible
- `int` or `float` → `double`
  - Exact conversion (32-bit ints; 52-bit frac + 1-bit sign)
- `long int` → `double`
  - Depends on word size (32-bit is exact, 64-bit may be rounded)
- `double` or `float` → `int`
  - Truncates fractional part (rounded toward zero)
  - E.g. 1.999 -> 1, -1.99 -> -1

Demo: Casting

Aside: Computerphile

An amazing YouTube channel with great Computer Science explanations.

If today's material was confusing at all, their 2's complement and floating point videos are fantastic!

Ex14 due Monday, HW5 due Sunday!

Ex14 is due before the beginning of the next lecture

- Link available on the website: https://courses.cs.washington.edu/courses/cse374/24wi/exercises/

HW5 due Sunday 11.59pm!

- Instructions on course website: https://courses.cs.washington.edu/courses/cse374/24wi/homeworks/hw5/