CSE 374 Programming Concepts and Tools

Lecture 15 - Variable Types and Storage

This Week

Today: Variable Types and Storage

Next time: Buffer overflow, Memory Architecture

Today

Revisiting Data

Characters: `char`

Integers: int

- Unsigned
- Signed: Two's Complement

Floating Point Numbers: `float`, `double`

Conversions

Review: Numbers Can Represent Anything

Text files

- ASCII: uses one byte to represent a single character for 2^7=128 different characters; 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, 한, 😀 (2+ bytes to represent)

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

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

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.

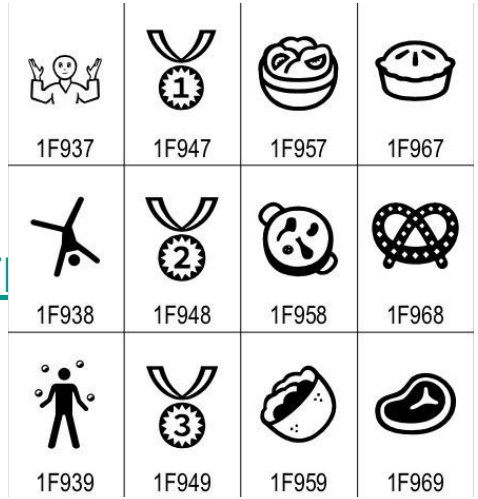
Each number corresponds to a different character, such as uppercase and lowercase letters, digits, and common symbols.

- e.g. 65 = 'A', 66 = 'B', 97 = 'a', 98 = '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 (UTI

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

Bonus Challenge

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

Question: Given a 25 character 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 25 character 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 25 character 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?

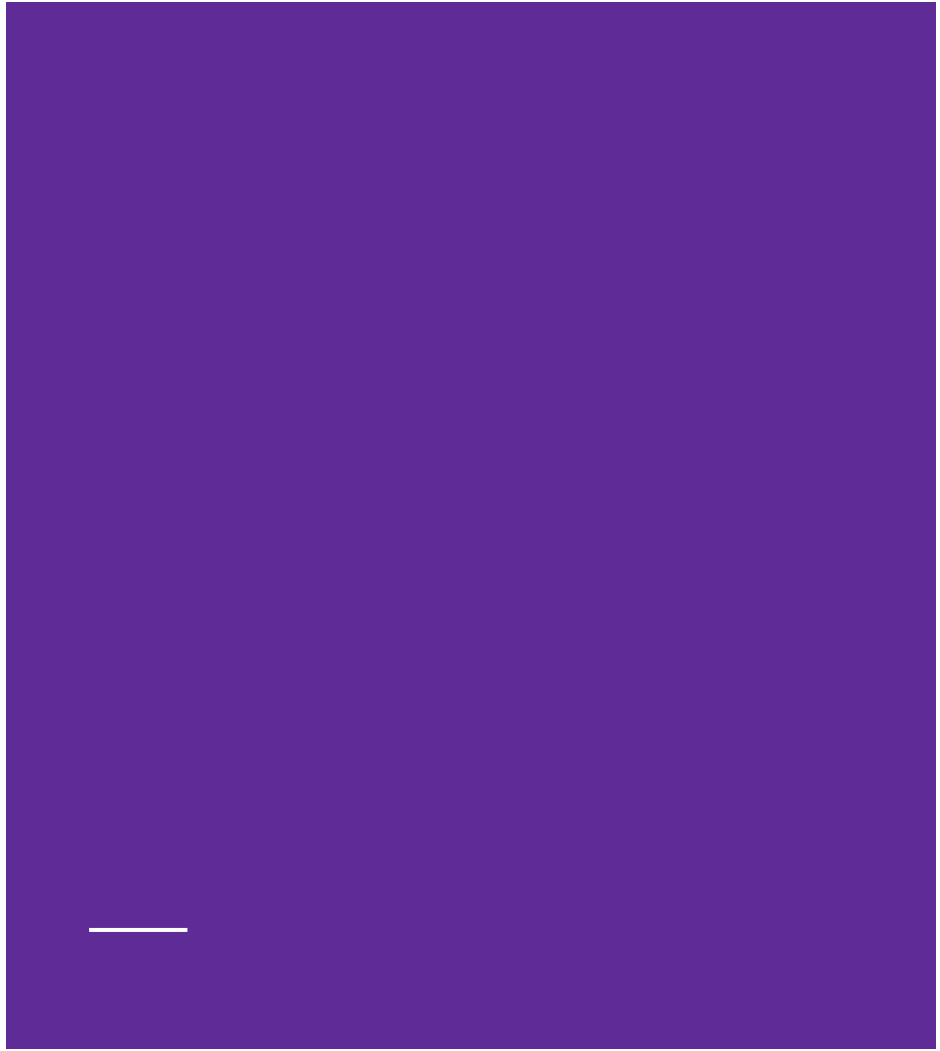
Answer: Sum the ASCII values of characters contained in the string, then subtract it from the *total* value of the ASCII alphabet.

```
sum("abcdefghijklmnopqrstuvwxyz") == 2847

sum("abdefghijklmnopqrstuvwxyz") == 2748

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

Integers



Review: Number systems and BASE

Generally use base 10

(decimal)

374

$3\times100 + 7\times10 + 4\times1$

$3\times10^2 + 7\times10^1 + 4\times10^0$

Digital systems - base 2

(binary)

374 = **0b**101110110

$1\times2^8 + 0\times2^7 + 1\times2^6 + 1\times2^5 + 1\times2^4 + 0\times2^3 + 1\times2^2 + 1\times2^1 + 0\times2^0$

Base 16 - very compact

(hexadecimal)

374 = **0x**176

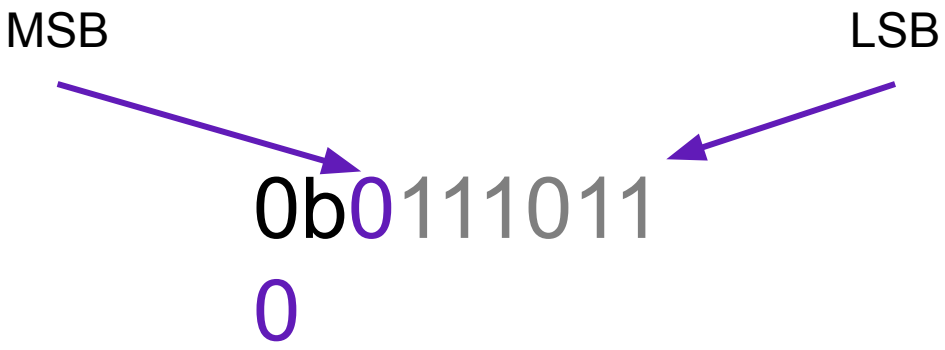
$1\times16^2 + 7\times16^1 + 6\times16^0$

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.

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.



How do we do arithmetic in binary again?

Very similarly to in decimal:

- Starting with the *least* significant bit and moving *left*ward, add the numbers in each place. If they add to 0 or 1, then you don't have to carry anything
- If they add to 2 = 0b10, then you write a 0 for that place and then carry a 1(0) to the next column!

Some examples on the board:

- 0b**0001** + 0b**0010** = …
- 0b**0100** + 0b**0101** = …
- 0b**0011** - 0b**0001** = …
- 0b**0101** - 0b**0010** = …

Exercise: Arithmetic in Binary

Try to calculate the following in binary on your own:

- x = 0b**0001** + 0b**0100**
- y = 0b**0010** + 0b**0010**
- a = 0b**0111** - 0b**0101**
- b = 0b**1011** - 0b**0100**

Then share with a neighbor.

Integer representations

The hardware (and C) supports two flavors of integers

- unsigned – only the non-negatives, follow standard base 2 system
  - We've been treating most binary/hex values as unsigned so far
- signed – both negatives and non-negatives
  - Signed numbers are stored 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): $2^4-1 \rightarrow 1111 \rightarrow 2^3+2^2+2^1+2^0 \rightarrow 8+4+2+1 \rightarrow 15$
- 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;`)

How to represent signed integer?

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

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

Not used in practice for integers

Benefits:

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

Examples (8 bits):

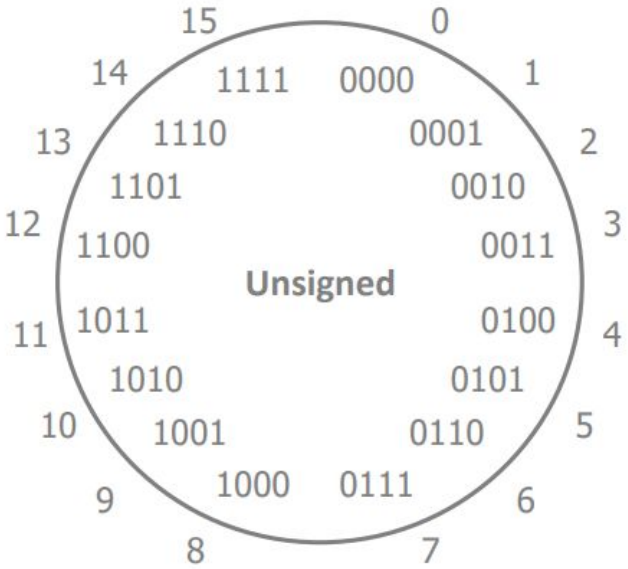
- 0x00 = 0b00000000 is non-negative, because the sign bit is 0
- 0x7F = 0b01111111 is non-negative (+127)
- 0x85 = 0b10000101 is negative (-5)
- 0x80 = 0b10000000 is negative … 0?

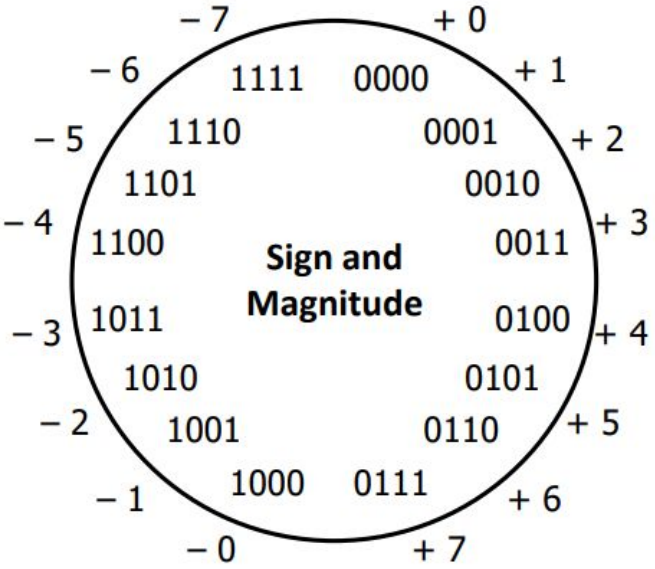
Sign and Magnitude

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

Drawbacks?

Not used in practice for integers



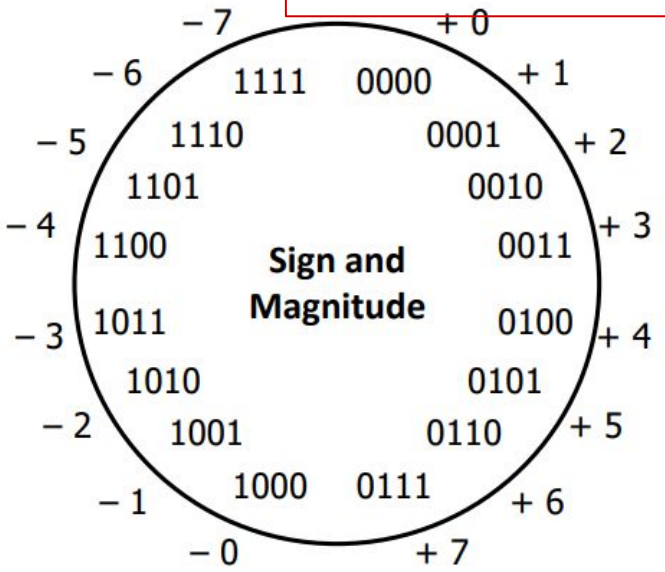


Sign and Magnitude

Drawbacks?

- Two representations of **0** (bad for checking equality)
- What about arithmetic?
  - Is 4 - 3 = 4 + (-3)?

Not used in practice for integers

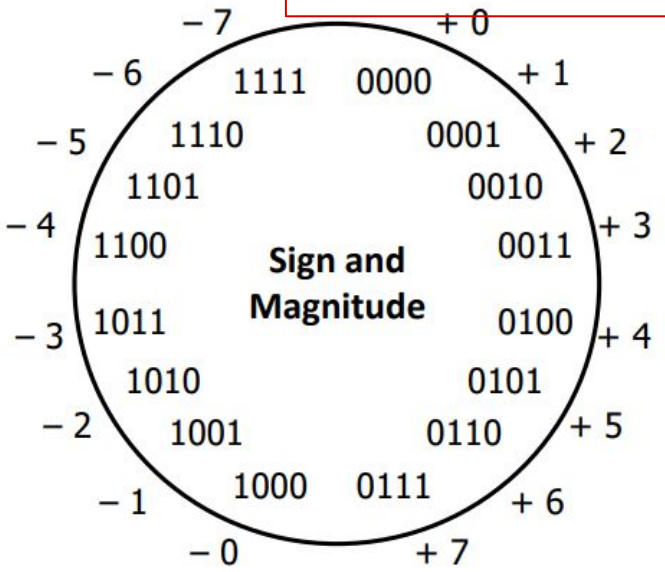


Sign and Magnitude

Drawbacks?

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

Not used in practice for integers



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:

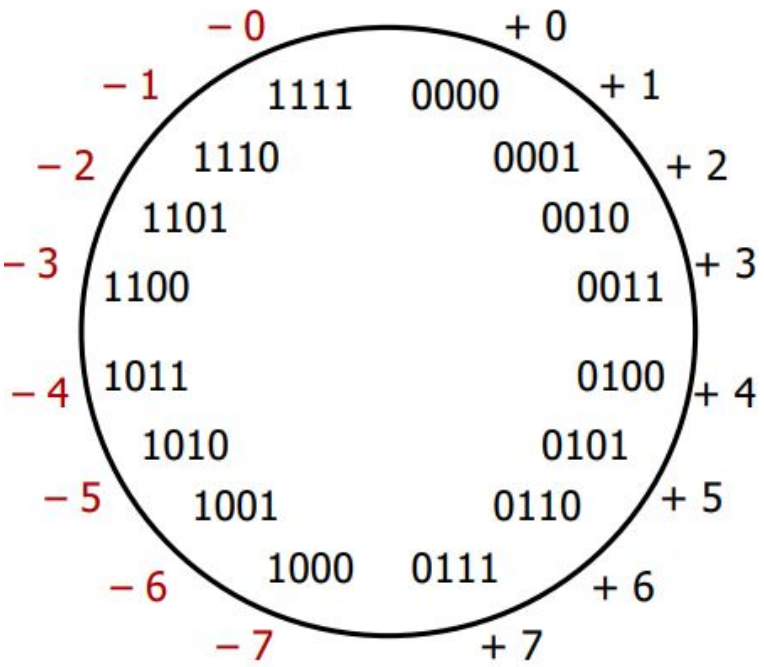
- "Flip" negative encodings so incrementing works
  - Basically, for each 0, make it a 1, and each 1, make it a 0

0101 (5) → 1010 (-5)

0111 (7) → 1000 (-7)

0000 (0) → 1111 (...-0?)

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



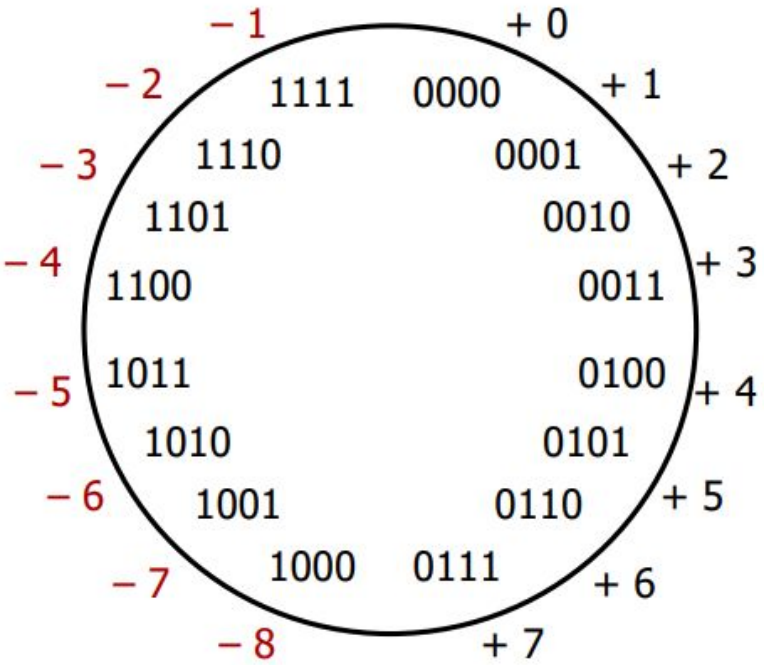
Two's Complement

Let's fix these problems:

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

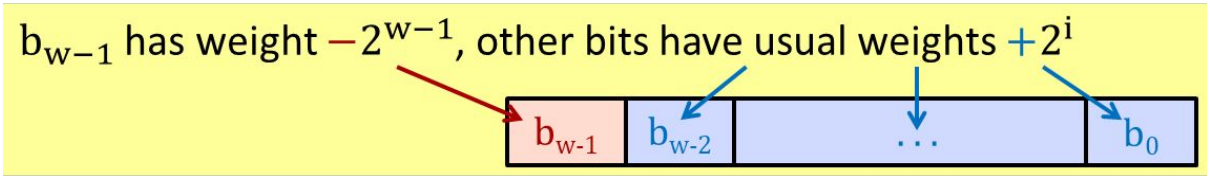
MSB **still** indicates sign!

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

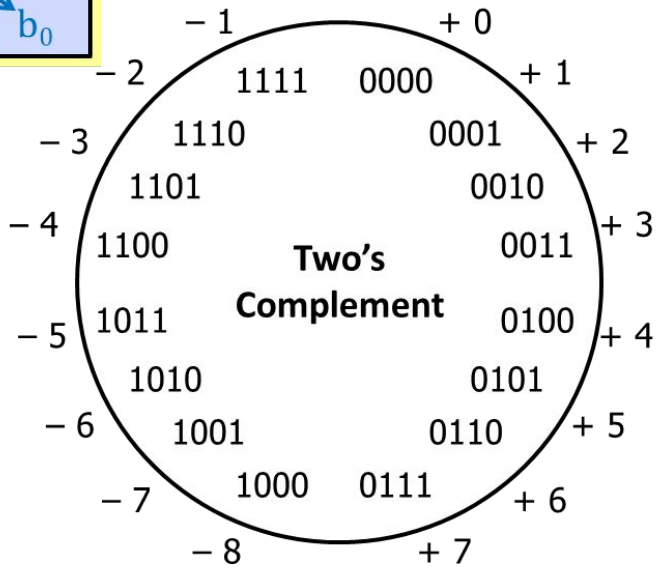


Two's Complement Negatives

Accomplished with one neat mathematical trick!

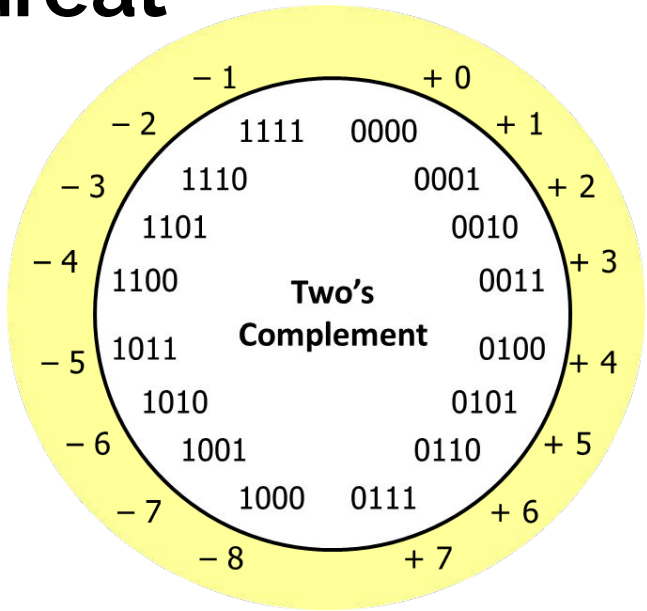


- 4-bit Examples:
  - 1010_2 unsigned: $1\times2^3+0\times2^2+1\times2^1+0\times2^0$ = **10**
  - 1010_2 two's complement: $-1\times2^3+0\times2^2+1\times2^1+0\times2^0$ = **–6**
- -1 represented as: $1111_2 = -2^3+(1\times2^3+1\times2^1+1\times2^0)$
- MSB makes it super negative, add up all the



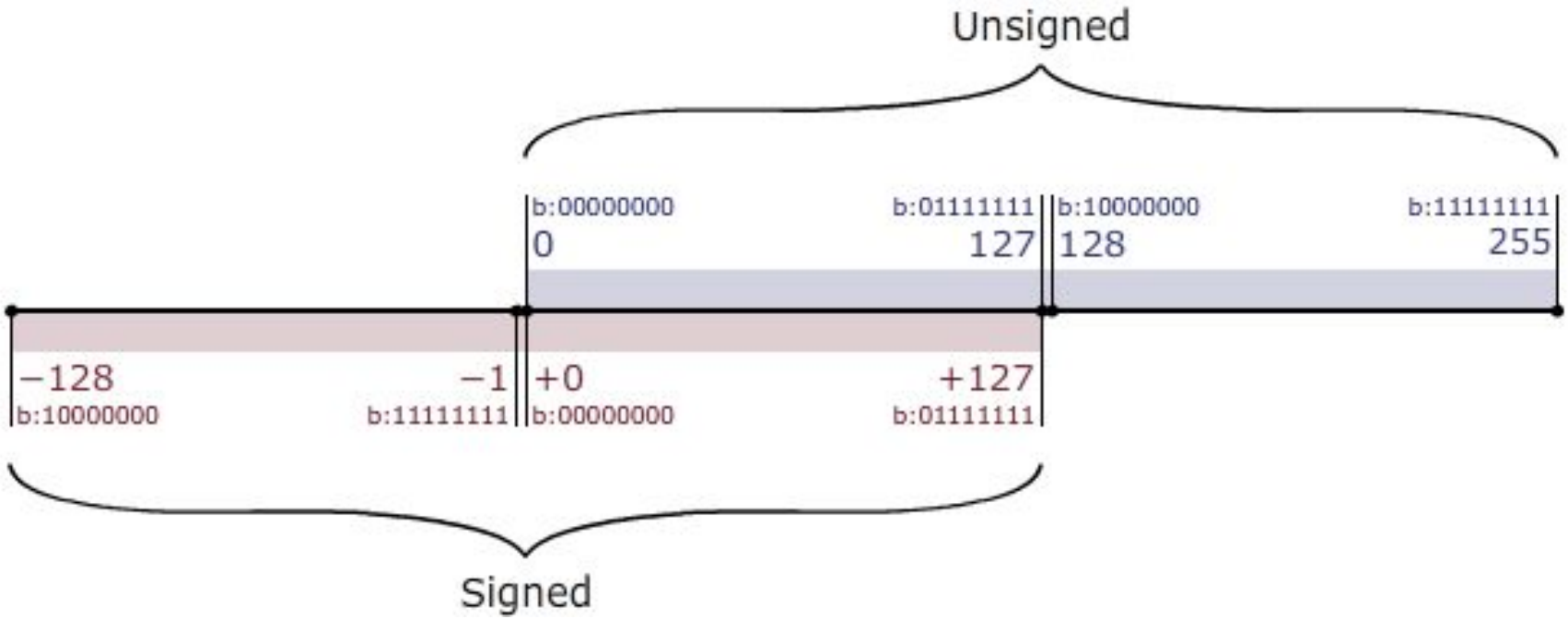
Why Two's Complement is So Great

- 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
```

Example: 8-bit integers



Polling Time!

Please answer in the LP15 assignment on Gradescope!

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

Which of the following numbers is NOT a valid interpretation of `x` using any of following number representation schemes discussed today?
(Unsigned, Sign and Magnitude, Two's Complement)

- -4
- -3
- -5
- 11

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: $1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 = 8 + 2 + 1 =$ **11**

Sign and magnitude: $-(0\times2^2 + 1\times2^1 + 1) = -(2 + 1) =$ **-3**

Two's complement: $-1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 = -8 + 2 + 1 =$ **-5**

Now, arithmetic completely works!

Suppose we have 4 bit 2's complement signed integers

0b0101 = 5

We can add 1, and get 0b0110 = 6 😎

We can add 1 again, and get 0b0111 = 7 😎

We can add 1 again, and get 0b1000 = …-8? 😨

And if we subtract 1 from that, we get 0b0111 = 7?! 😱

What just happened?! I thought that arithmetic was supposed to work in 2's complement 🥺

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’s-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 and unsigned:

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

Implicit casting also occurs via assignments and function calls:

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

Avoiding Gotchas in C with (Un)signed Ints

C doesn't dictate the int representation method – the compiler does

If you want warnings for implicit casts, **use `-Wsign-conversion`**

- `-Wall` doesn't produce warnings!

All constants in C are assumed to be signed unless you append a U at the end of a number, e.g., **`15U`** is the unsigned integer 15

Type promotion: Implicitly casting from small data types to larger ones, e.g. int to long

⚠️ If a larger type is cast from a smaller one, the values will change! ⚠️

- How they change depends on the types involved

Floating Point Numbers

Real numbers (e.g. 3.14159) (...kinda)

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

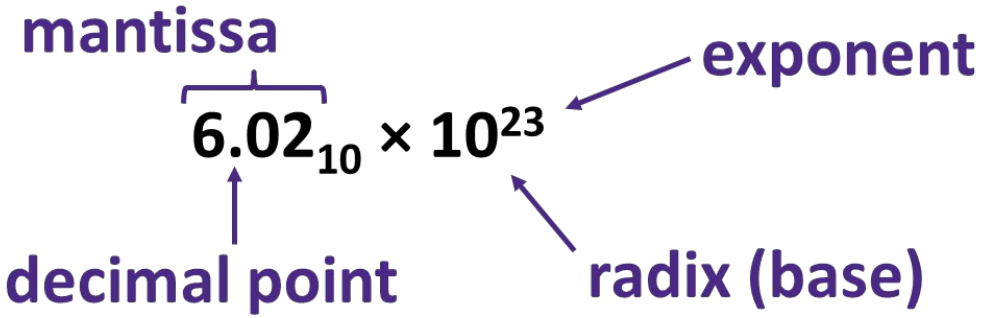
- i.e. No decimal point: 42 vs 1.5
- Remember `int` vs `double` from Java?

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

Floats are stored kind of like scientific notation numbers

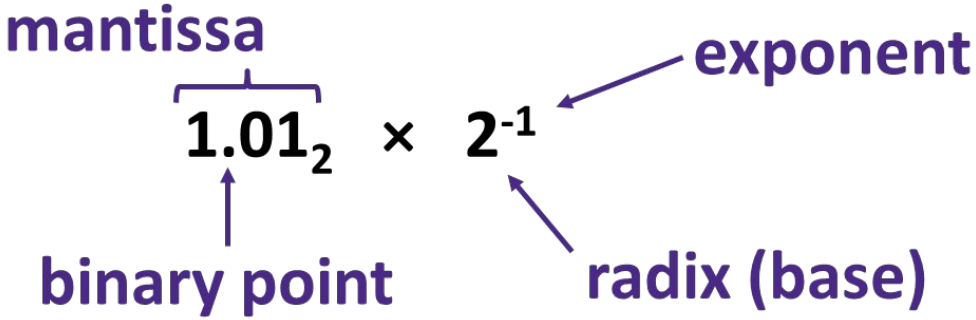
- e.g. $0.123 = 1.23 \times 10^{-1}$

Scientific Notation (Decimal)



- *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)



- Computer arithmetic that supports this is called floating point due to the "floating" of the binary point
  - The binary point could eventually appear anywhere when multiplied out
- Declare such a variable in C as `float` (or `double`)

Scientific Notation Translation

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

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 Standard

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

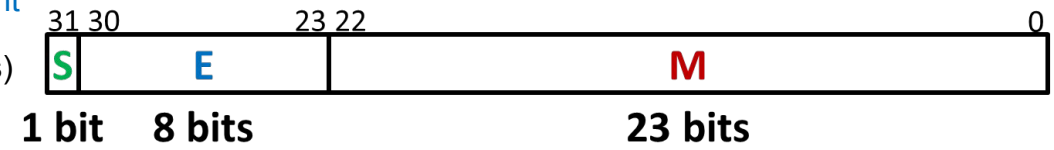
Driven by numerical concerns

- **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: Biased Notation

Use biased notation

- Read exponent as unsigned, but with *bias* of $2^{w-1}-1$ = 127
- Representable exponents roughly ½ positive and ½ negative
- Example: An exponent of 0 (Exp = 0) is represented as E = 0b 0111 1111
  - The bias is subtracted from the representation
  - So, 0b 0111 1111 - 127 = 0b 0111 1111 - 0b 0111 1111 = 0

Why biased?

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

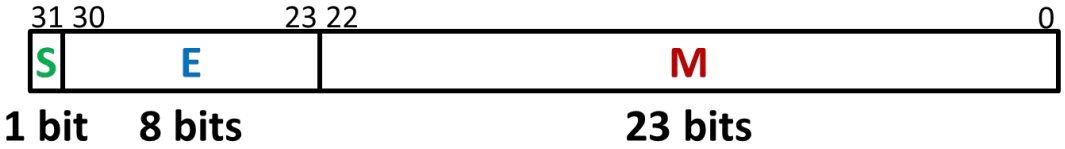
Practice: To encode in biased notation, *add* the bias then encode as an unsigned number:

- Exp = 1 → 128 → E = 0b 1000 0000
- Exp = -63 → 64 → E = 0b 0100 0000

To decode biased notation, subtract the bias:

- E = 0b 1111 1111 → 255 - 127 → 128

The Mantissa (Fraction) Field



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

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

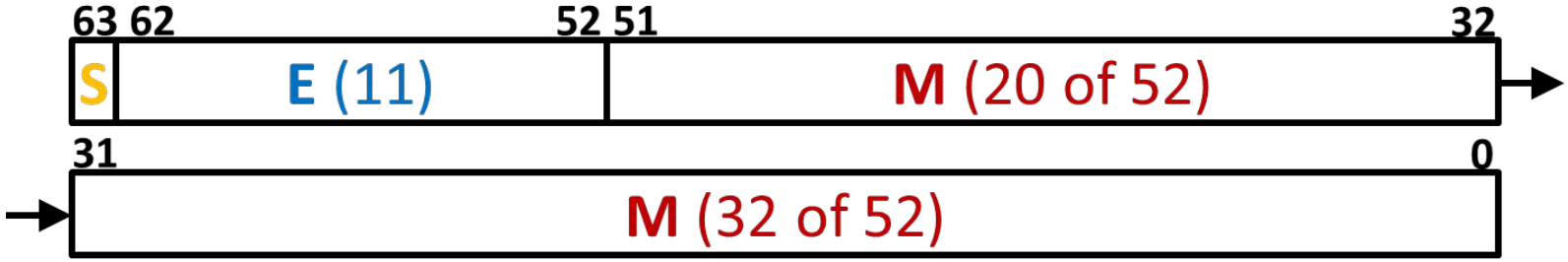
- Example: 0b 0011 1111 1100 0000 0000 0000 0000 0000 is read as $1.1_2 = 1.5_{10}$, *not* $0.1_2 = 0.5_{10}$
- Gives us an extra bit of ***precision***

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** 😭

- So a + (b + c) may not equal (a + b) + c, or a * (b + c) may not equal a * b + a * c
- 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 floating point numbers, 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!

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

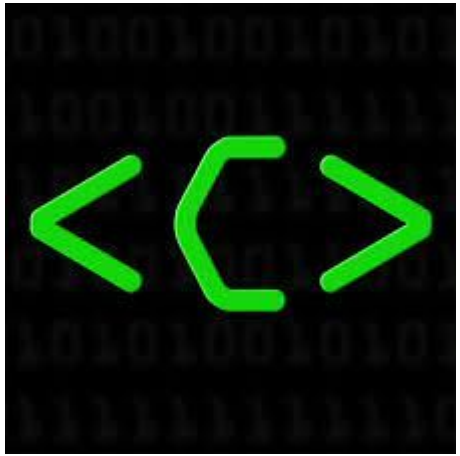
Demo: Casting

Aside: Computerphile

An amazing YouTube channel with great Computer Science explanations.

Check out their 2's complement and floating point videos!

- Lots of nitty gritty we didn't get into to today



Assignment Reminders

EX 13 released, due before class on Friday