Data Representation

1

CS/COE 0449
Introduction to Systems Software

Luis Oliveira
(with content borrowed from wilkie and Vinicius Petrucci)

Positional Numbers

A quick review!

- The numbers we use are written **positionally**: the position of a digit within the number has a meaning.

$$2\ 0\ 2\ 4 = \begin{array}{r} 2\,0\,0\,0 \\ 0\,0\,0 \\ 2\,0 \\ +\quad 4 \end{array} = \begin{array}{l} 2 \times 10^3 \\ 0 \times 10^2 \\ 2 \times 10^1 \\ 4 \times 10^0 \end{array}$$

What are the limits?

Using base **10**

- A 4-digit number, e.g.:

$$2024$$

- Has the value

$$2 \times \mathbf{10}^3 + 0 \times \mathbf{10}^2 + 2 \times \mathbf{10}^1 + 4 \times \mathbf{10}^0$$

- Using 4 digits we can represent 10000 different numbers
- The smallest non-negative number representable with 4 digits is 0
- The largest number representable with 4 digits is 9999, or $10^4 - 1$
- Using 10 symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

$\boxed{\text{NOT 6s!!}}$

A base-10 number system

Using base **10**

- A number represented by the digits

$$d_{n-1} \dots d_1 d_0$$

- Has the value

$$d_{n-1} \times \mathbf{10}^{n-1} + \cdots + d_1 \times \mathbf{10}^1 + d_0 \times \mathbf{10}^0$$

- Using n digits we can represent $\mathbf{10}^n$ different numbers

- The smallest non-negative number representable with n digits is 0

- The largest number representable with n digits is $\mathbf{10}^n - 1$

- Using **10** symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Binary (base-2)

- We call a **B**inary dig**IT** a **bit** – a single 1 or 0
- When we say an *n*-bit number, we mean one with *n* binary digits

MSB							LSB
1	0	0	1	0	1	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128s	64s	32s	16s	8s	4s	2s	1s

1001 0110 =

$1 \times 128 +$
$0 \times 64 +$
$0 \times 32 +$
$1 \times 16 +$
$0 \times 8 +$
$1 \times 4 +$
$1 \times 2 +$
0×1

$= 150_{10}$

To convert binary to decimal: ignore 0s, add up place values wherever you see a 1.

It's the only odd number!

Let's make a base-2 number system

Using base **2**

- An 4-digit number, e.g.:

1011

- Has the value

$$1 \times \mathbf{2}^3 + 0 \times \mathbf{2}^2 + 1 \times \mathbf{2}^1 + 1 \times \mathbf{2}^0$$

- Using 4 digits we can represent ______ different numbers
- The smallest non-negative number representable with 4 digits is ______
- The largest number representable with 4 digits is _______, or 2—_____
- Using ___ symbols: ___________________________

Let's make a base-2 number system

Using base **2**

- A number represented by the digits

$$d_{n-1} \ldots d_1 d_0$$

- Has the value

$$d_{n-1} \times \mathbf{2}^{n-1} + \cdots + d_1 \times \mathbf{2}^1 + d_0 \times \mathbf{2}^0$$

- Using n digits we can represent $\mathbf{2}^n$ different numbers

- The smallest non-negative number representable with n digits is 0

- The largest number representable with n digits is $\mathbf{2}^n - 1$

- Using **2** symbols: 0, 1

Hexadecimal, or "hex" (base-16)

- Digit symbols after 9 are A-F, meaning 10-15 respectively.
- Usually we call one hexadecimal digit a *hex digit*. No fancy name :(

$$\underset{16^7}{0}\ \underset{16^6}{0}\ \underset{16^5}{3}\ \underset{16^4}{B}\quad \underset{16^3}{E}\ \underset{16^2}{E}\ \underset{16^1}{7}\ \underset{16^0}{0} =$$

$$0 \times 16^7 + 0 \times 16^6 + 3 \times 16^5 + 11 \times 16^4 + 14 \times 16^3 + 14 \times 16^2 + 7 \times 16^1 + 0 \times 16^0 = 3{,}927{,}664_{10}$$

- Hex is usually short-hand for binary… For example,
  - Instead of writing 00000000001110111110111001110000
  - We write: 003BEE70
  - log2(16) = 4 ➔ What does this mean? ☺

Why is hex short-hand for binary?

Convert in groups of 4 bits from binary <-> hex

0010	0100	0110	1111	1000	0110	1001	0101
2	4	6	F	8	6	9	5

$246F8695_{16}$

0x246F8695

What I expect you to know?

- Correctly interpret a positional base-b number. (e.g. base-2/16)
  - If you are taking CS447 now, this should be taught soon!

- Convert between any combination of:
  - Base-2 (binary)
  - Base-10 (decimal)
  - Base-16 (hexadecimal)

Sun	Mon	Tue	Wed	Thu	Fri	Sat
1/8	**Introduction** 1/9	1/10	Numerical Representation and Unsigned Integers 1/11	1/12	1/13	1/14
1/15	MLK Day 1/16	1/17	Signed Integers, Extension, Truncation, and Addition 1/18	1/19	Lab0 due 1/20	1/21
1/22	MIPS Instructions, Registers, and Math 1/23	1/24	Memory and Addresses 1/25	1/26	Lab1 due 1/27	1/28

- If you do not know this, review my CS0447 slides:
  - Check Bonus Slides at the end of this presentation!
  - Or go to: https://cs0447.gitlab.io/sp2024/schedule
  - Fun little exercise: https://cs0447.gitlab.io/sp2024/exercises/ex1/welcome
  - Come to office hours!

Think about what you know about number bases

Which were your pain-points?
- Concepts you struggle/d with
- Concepts your friends struggle/d with

Any advice for students taking 447 right now?

I'll ask you to share in 1 minute ☺

E.g.:
- Converting between bases
- Reading numbers
- Hexadecimal

Integer Encoding

Storing data bit by bit

The finitude of variables

These slides were made for high-school students and their parents!

But I hate wasting slides!

Let's program – this is C++ for reasons

I made a cute tiny positive number because computers work with much larger numbers.

Which are terrible as examples ☺

```cpp
int main() {

    positive_tiny number = 0;
    print << (number) << enter;

    return 0;
}
```


```cpp
int main() {

    positive_tiny number = 2;
    print << (number) << enter;

    return 0;
}
```


```cpp
int main() {

    positive_tiny number = 14;
    print << (number) << enter;

    return 0;
}
```

Let’s program

```
int main() {

    positive_tiny number = 14;
    print << (number+1) << enter;

    return 0;
}
```

```
15
```

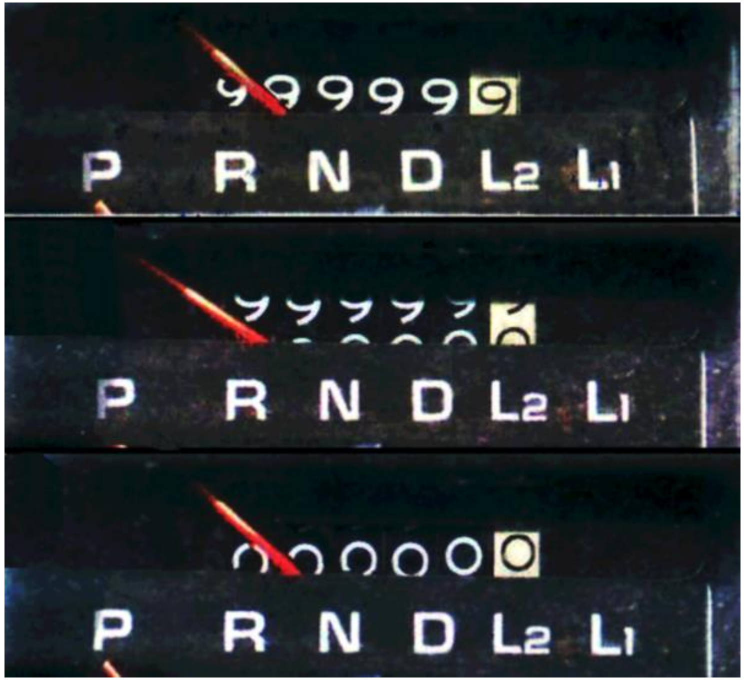
```
int main() {

    positive_tiny number = 14;
    print << (number+2) << enter;

    return 0;
}
```

```
0
```

```
int main() {

    positive_tiny number = 14;
    print << (number+3) << enter;

    return 0;
}
```

```
1
```

What’s up, Doc? 🥕
– Overflow!



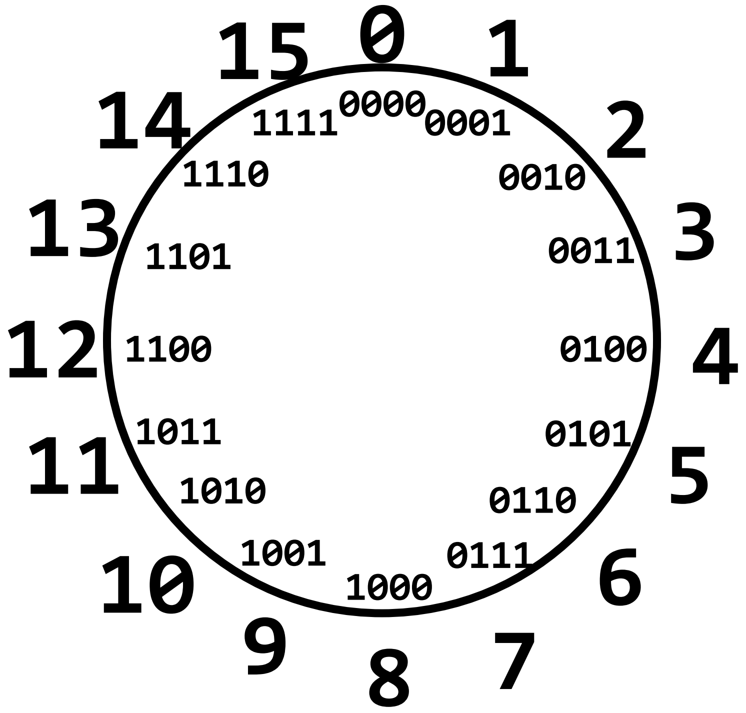
By Hellbus - Own work, Public Domain,
https://commons.wikimedia.org/w/index.php?curid=3089111

14	1110

positive_tiny

Can only hold 4 bits

It's more like a carrousel



0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

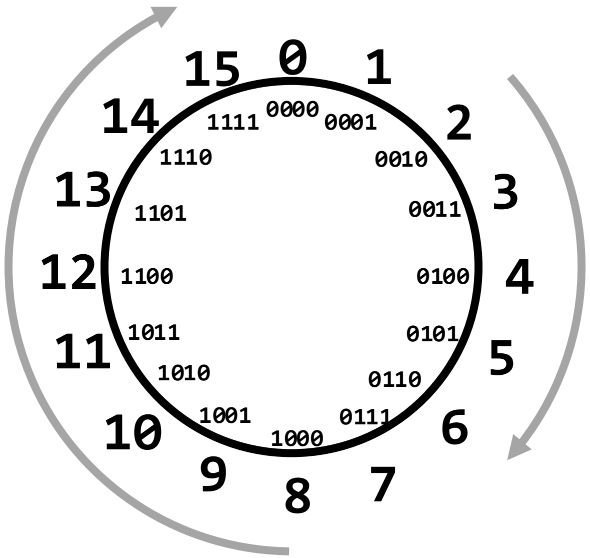
Computer Maths is always right

```
positive_tiny number = 8;
print << (number+7) << enter;
```

```
15
```

```
positive_tiny number = 2;
print << (number+4) << enter;
```

```
6
```



Computer Maths is ~~always~~ isn't right

```
positive_tiny number = 8;
print << (number+7) << enter;
```

```
15
```

```
positive_tiny number = 2;
print << (number+4) << enter;
```

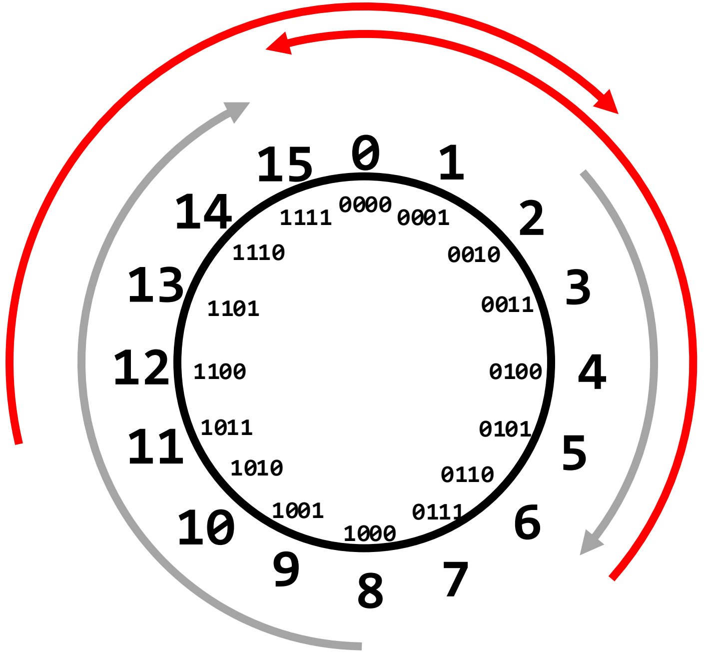
```
6
```

```
positive_tiny number = 6;
print << (number-7) << enter;
```

```
15
```

```
positive_tiny number = 11;
print << (number+7) << enter;
```

```
2
```



This is fine!!!!

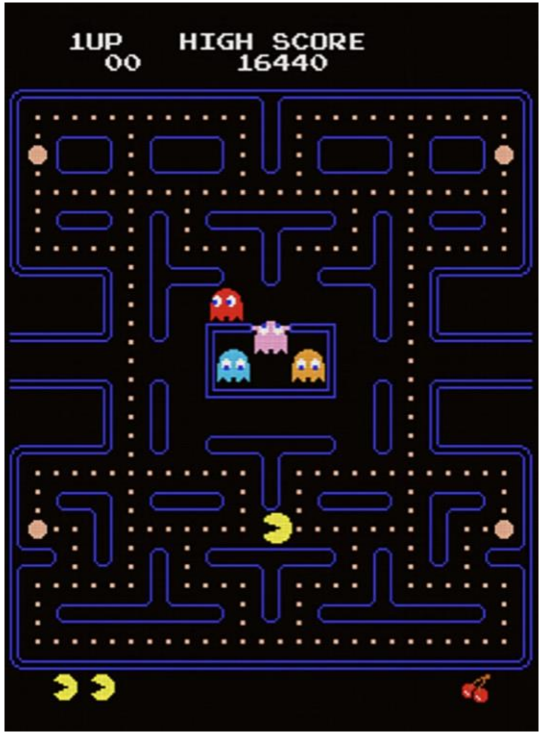
After all, actual computer numbers can hold MUCH bigger values

Pac-Man

– I know this is not exactly how it happens! But it's the same process!

Level: 255

Binary: 1 1 1 1 1 1 1 1



Source: https://www.cnn.com/style/article/pac-man-40-anniversary-history/index.html

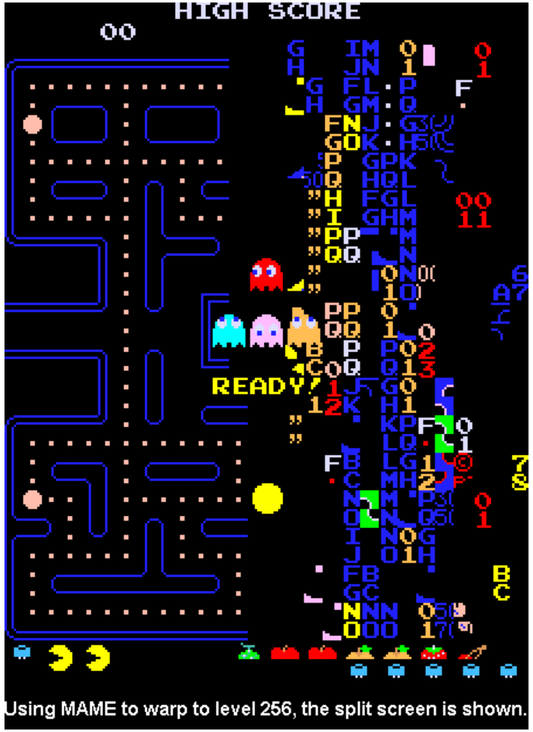
+ 1 = 1 0 0 0 0 0 0 0 0

Bitbucket

Congrats, you made it to computer chaos

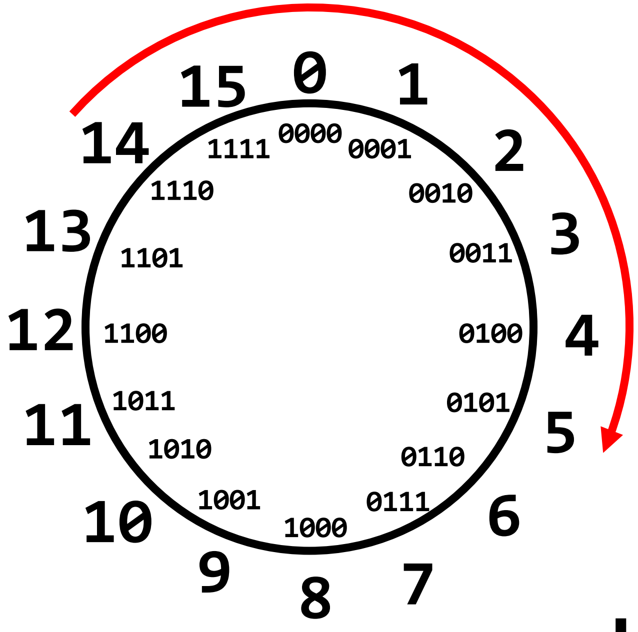
Level: 256

Binary:



Source: https://pacman.fandom.com/wiki/Map_256_Glitch

Positive number overflow?




```
  1110
 +0111
 -----
 10101
```

what is 14 + 7?

If the result is smaller than either addend, there is an overflow

Because there is no space for the extra carry

What about negative numbers?

Let’s program

I made a cute tiny number that now holds both positive and negative numbers

```
int main() {

    tiny number = -6;
    print << (number-1) << enter;

    return 0;
}
```

```
-7
```

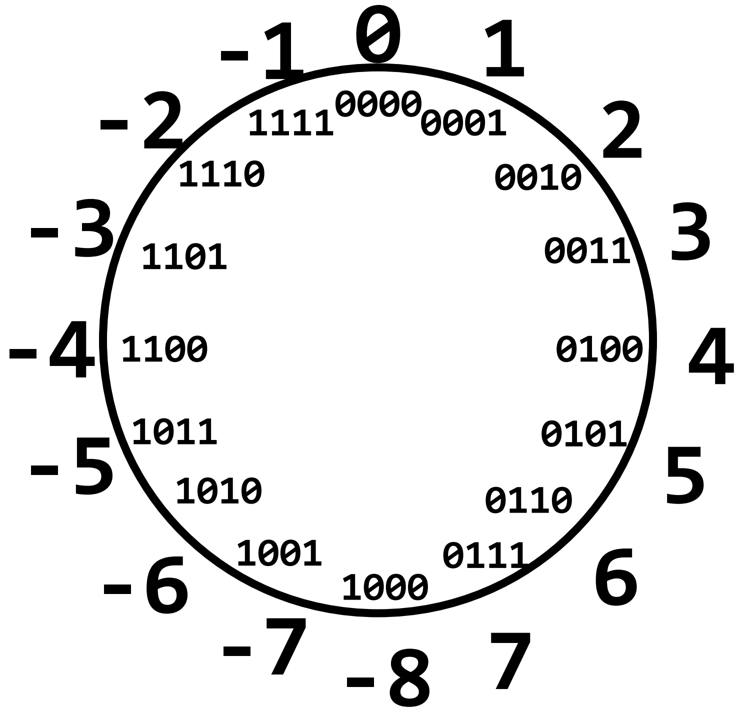
```
int main() {

    tiny number = -7;
    print << (number-1) << enter;

    return 0;
}
```

```
-8
```

```
int main() {

    tiny number = -8;
    print << (number-1) << enter;

    return 0;
}
```

```
7
```

Remember the carrousel



0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111

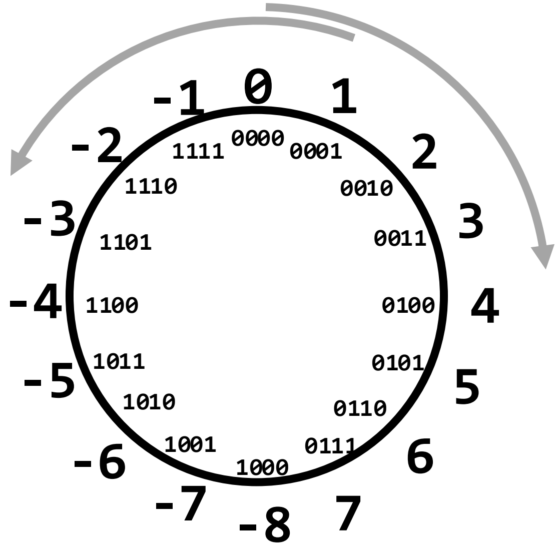
Computer Maths is always right

```
tiny number = 1;
print << (number-4) << enter;
```

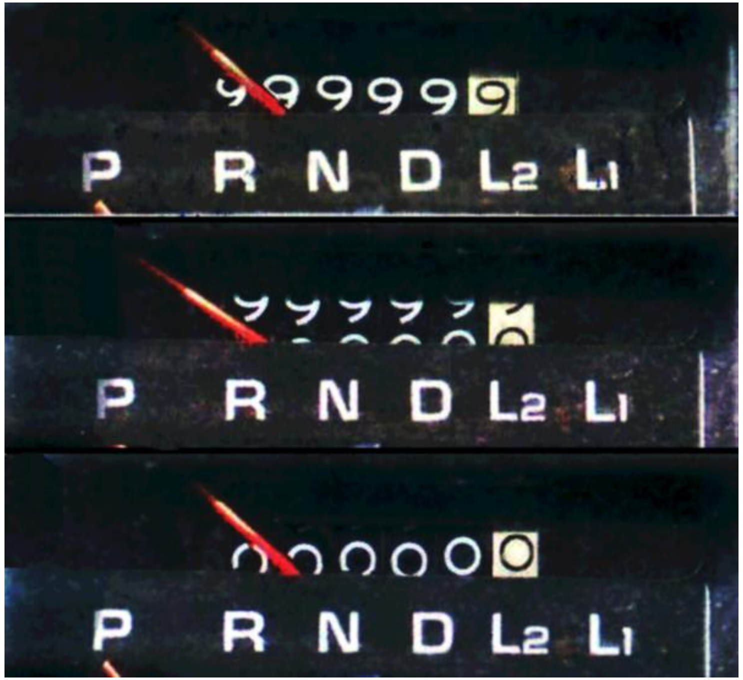
```
-3
>
```

```
tiny number = 0;
print << (number+4) << enter;
```

```
4
>
```



What’s up, Doc?



By Hellbus - Own work, Public Domain,
https://commons.wikimedia.org/w/index.php?curid=3089111

6	0110

Computer Maths is always right

isn't

```
tiny number = 1;
print << (number-4) << enter;
```

```
-3
```

```
tiny number = 0;
print << (number+4) << enter;
```

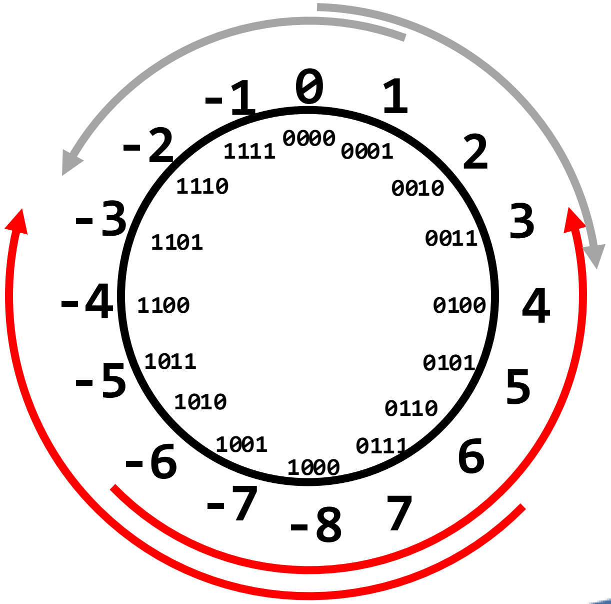
```
4
```

```
tiny number = -6;
print << (number-7) << enter;
```

```
3
```

```
tiny number = 6;
print << (number+7) << enter;
```

```
-3
```



This is fine!!!!

After all, actual computer numbers can hold MUCH bigger values

Boeing

https://www.govinfo.gov/content/pkg/FR-2015-05-01/pdf/2015-10066.pdf

AUTHENTICATED U.S. GOVERNMENT INFORMATION GPO

Federal Register / Vol. 80, No. 84 / Friday, May 1, 2015 / Rules and Regulations **24789**

deactivation on Model 787 airplanes. This AD was prompted by the determination that a Model 787 airplane that has been powered continuously for 248 days can lose all alternating current (AC) electrical power due to the generator control units (GCUs) simultaneously going into failsafe mode. This condition is caused by a software counter internal to the GCUs that will overflow after 248 days of continuous power. We are issuing this AD to prevent loss of all AC electrical power,

By pjs2005 from Hampshire, UK - Boeing 787 N1015B ANA Airlines, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=71147495

Can we detect that?

How much is -6 - 7?

```
 1010
+1001
 0011
```

Add 2 negative numbers, the result is positive ❌

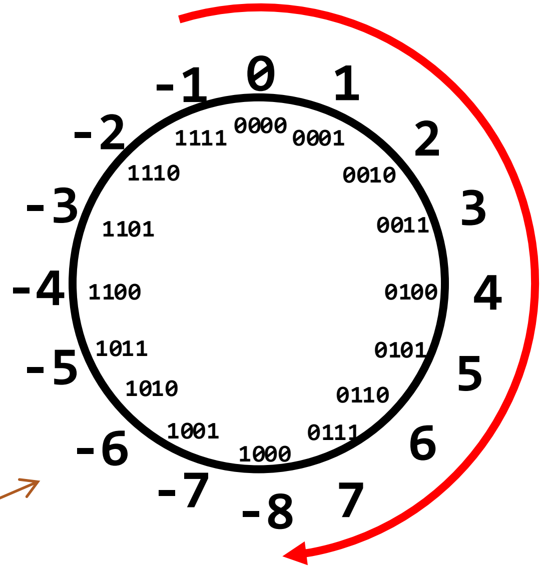
How much is 6 + 7?

```
 0110
+0111
 1101
```

Add 2 positive numbers, the result is negative ❌

What about this?

How can we detect if operations with different signs overflow?



This is impossible: Max positive = 7
-1+7=6!

What if a language ($C_{off} \ldots C_{off}$) had ONLY positive variables?

With the person next to you discuss:

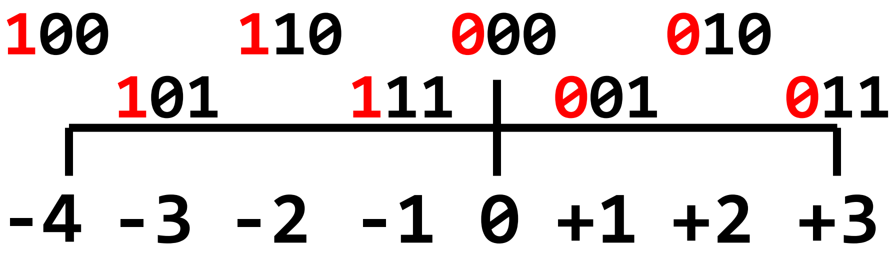
- When would you use them? (to store which kind of data)
- What issues do you foresee?

I'll ask you questions about your findings after 1 minute

The quirks of signed numbers

Signed Numbers (2's Complement)

- Representing negative numbers.
  - But it's a little strange!
- Hmm, it's a little lopsided: -4 **doesn't have a valid positive number**.



- If it's positive, then I can clearly see how much it's worth!
- I can tell it's negative if it starts with a 1 ☺
  - But how exactly… can we tell the value of a negative number?

Formally – don't need to memorize these formulas

- Encode/Decode 2's complement:
  - Given *n*-bit number *x*

$$\text{Encode}(x) = \begin{cases} x, if\ x \geq 0 \\ x + 2^n, if\ x < 0 \end{cases}$$

Assuming 4-bits
$\text{Encode}(3) = 3 \Rightarrow 0b0011$
$\text{Encode}(-3) = -3 + 2^4 = 13 = 0b1101$

$$\text{Decode}(x) = \begin{cases} x, if\ x < 2^{n-1} \\ x - 2^n, if\ x \geq 2^{n-1} \end{cases}$$

$$\text{Decode}(x) = -x_{n-1} \cdot 2^{n-1} + \sum_{k=0}^{n-2} x_k \cdot 2^k$$

Assuming 4-bits

$$\text{Decode}(-3) = -1 \cdot 2^{4-1} + \sum_{k=0}^{4-2} x_k \cdot 2^k$$

$$= -\mathbf{1} \cdot 2^3 + \mathbf{1} \cdot 2^2 + \mathbf{0} \cdot 2^1 + \mathbf{1} \cdot 2^0$$

$$= -8 + 4 + 0 + 1 = -3$$

Two's complement arithmetic

- Negation

-(3) → 0011 → 1100 → 1101

bit pattern for positive 3? *flip!* *Add 1!*

-(-3) → 1101 → 0010 → 0011

bit pattern for negative 3? *flip!* *Add 1!*

- You don't need to subtract!!
  - flip(k)+1 == flip(k-1)
    - If you ignore the carry! ☺

Two's complement addition

- the great thing is: you can add numbers of either sign **without having to do anything special!**

to binary?

```
         0011
  3  →    0011
+ 9  →   +1001
 12       1100
```

bit pattern for -7... positive 7?

```
   3
+-7
  -4
```

0111

flip!

1000

+1

Ignore the carry

```
 0011
  0011
 +1001
  1100
```

this is negative, so what is it? flip!

0011

+1

0100

to decimal?

-4

the actual patterns of bits are the same.
so how does the computer "know" whether it's doing signed or unsigned addition?

Integer Ranges

- Recall:
  - The range of an unsigned integer is 0 to $2^n - 1$
  - **Q**: Why do we subtract 1?

- What is the range of a 2's complement number?
  - Consider the sign bit, how many negative integers?
  - Consider, now, the positive integers.
  - Remember 0.

$$-2^{n-1} \text{ to } 2^{n-1} - 1$$

Absolutely Bonkers

```java
public class AbsTest {
  public static int abs(int x) {
    if (x < 0) {
      x = -x;
    }

    return x;
  }

  public static void main(String[] args) {
    System.out.println(
      String.format("|%d| = %d", Integer.MIN_VALUE, AbsTest.abs(Integer.MIN_VALUE))
    );
  }
}
// Outputs: |-2147483648| = -2147483648
```

With the person next to you: Take 1 minute to think about what happens when you flip the sign of the smallest Negative 2’s complement number.

I’m going to ask someone to explain

Q: How many bits is a Java `int`? What happened here?

What I expect you to know?

- Define what a signed numbers are NOT negative! They CAN be negative!
- Convert from/to 2’s complement encoding and decimal:
  - E.g. -3 => 11111101 E.g. 3 => 00000011
  - E.g. 11111110 => -2 E.g. 01111110 => 126
- Interpret Base-16 numbers as shorthand for Base-2 numbers
  - E.g.: Which of the following signed 16-bit integers is negative?
    0xFFFF
    0x1234
- If you do not know this, review my CS0447 slides:
  - Wait for next week’s 447 lecture!
  - You can review my CS447 slides: Signed numbers and Overflow
  - https://luisfnqoliveira.gitlab.io/cs447_f2021/schedule
  - Come to office hours

Integers: Now I can C it

C what I did there?

Integers in Java

- Integers are signed variables using 2’s complement: -2^{n-1} to $2^{n-1} - 1$
  - Where “n” is determined by the variable’s size in bits.

- Integer Types:
  - `byte` 8 bits
  - `short` 16 bits
  - `int` 32 bits
  - `long` 64 bits

Integers in ~~Java~~ C

- C allows for variables to be declared as either signed or unsigned.
  - Remember: “signed” does not mean “negative” just that it *can* be negative.

- An unsigned integer variable has a range from 0 to $2^n - 1$
- And signed integers are usually 2’s complement: -2^{n-1} to $2^{n-1} - 1$
  - Where “n” is determined by the variable’s size in bits.
  - 2’s complement is <u>not mandated</u> by C… but most machines use it!
    - This is changing!

- Integer Types: (signed by default, their sizes are arbitrary!!)

`char`	`unsigned char`	`8 bits (byte)`
`short int`	`unsigned short int`	`16 bits (half-word)`
`int`	`unsigned int`	`32 bits (word)`
`long int`	`unsigned long int`	`64 bits (double-word)`

Yes! Char is an integer type!!

- Usually no strong reason to use anything other than (un)signed int.

Are these true or false? – trick question TM

- Given this declaration:

```
signed int x,y = ?;
unsigned int ux = ?;
```

- Mark these statements as either True/False for **ALL** possible values of **x, y** and **ux**

```
1. x < 0              ⇒    ((x*2) < 0)     _____
2. ux >= 0                                 _____
3. ux > -1                                 _____
4. x > y              ⇒    -x < -y         _____
5. x * x >= 0                              _____
6. x > 0 && y > 0     ⇒    x + y > 0       _____
7. x >= 0             ⇒    -x <= 0         _____
8. x <= 0             ⇒    -x >= 0         _____
```

✔ True
✘ False

Integers in C: Limits

Suggested HW:
Which other constants are defined in limits.h?

- Since sizes of integers are technically arbitrary…
  - They are usually based on the underlying architecture.
- … C provides standard library constants defining the ranges.
  - https://pubs.opengroup.org/onlinepubs/009695399/basedefs/limits.h.html

```c
#include <limits.h>               // Provides INT_MAX etc
#include <stdio.h>                // Provides printf

int main() {
  printf("%d ",  INT_MAX);        // Print the maximum signed int
  printf("%u\n", UINT_MAX);       // Print the maximum unsigned int
  return 0;
}               // Output: 2147483647 4294967295
```

Integer Sizes – sizeof

`sizeof` gives us the ability to programmatically obtain the byte size of data

```
sizeof(int); // -> 4 (=32b) on a typical 64-bit system

long long_variable = 0;
sizeof(long_variable); // ->8 (=64b) on a typical 64-bit system
```

Integer Sizes

```
#include <stdio.h>   // Gives us 'printf'

int main(void) {
  printf("sizeof(x):    (bytes)\n");
  printf("char:         %lu\n", sizeof(char));
  printf("short:        %lu\n", sizeof(short));
  printf("int:          %lu\n", sizeof(int));
  printf("unsigned int: %lu\n", sizeof(unsigned int));
  printf("long:         %lu\n", sizeof(long));
  printf("float:        %lu\n", sizeof(float));
  printf("double:       %lu\n", sizeof(double));
  return 0;
}
```

C data types – summary

Data Type	Typical 32-bit size	Typical 64-bit size	Typical 64-bit unsigned ranges	Typical 64-bit signed ranges
char	1	1	$0 .. 2^{8}-1$	$-2^{7} .. 2^{7}-1$
short	2	2	$0 .. 2^{16}-1$	$-2^{15} .. 2^{15}-1$
int	4	4	$0 .. 2^{32}-1$	$-2^{31} .. 2^{31}-1$
long	4	8	$0 .. 2^{64}-1$	$-2^{63} .. 2^{63}-1$
float	4	4	Erm….	Erm….
double	8	8	Erm….	Erm….
“address”	4	8	$0 .. 2^{64}-1$	Erm….

Integer Casting

Not Just a Witch or Wizard Thing
but it seems like it sometimes :/

Magic

- *Explicit* conversion between signed & unsigned

```
int sx, sy;
unsigned ux, uy;
sx = (int) ux;
uy = (unsigned) sy;
```

Casting

- C lets you move a value from an unsigned integer variable to a signed integer variable. (and vice versa)

- However, this is not always valid! Yet, it will do it anyway.
  - The binary value is the same, ***its interpretation is not***!
    - This is called ***coercion***, and this is a relatively simple case of it.
  - Since it ignores obvious invalid operations this is sometimes referred to as “weak” typing.
  - The strong/weak terminology has had very fragile definitions over the years and are arguably useless in our context. Let’s ignore them.

- Moving values between different types is called ***casting***
  - Which sounds magical and it sometimes is.

Example of coercion -> bits are unchanged

```c
#include <stdio.h>
#include <limits.h>
int main()
{
  int i = -1;
  unsigned u = i; // same as (unsigned)i;
  printf("i=%d, u=%u\n", i, u);
  return 0;
}
```

```
Output:
i=-1, u=4294967295
```

Careful!

- *Implicit* casting also occurs via assignments and procedure calls

```
sx = ux;
uy = sy;
```

```
int fun(unsigned u);
uy = fun(sx);
```

- Integer literals (constants)
  - By default, integer constants are considered *signed* integers
    - Hex constants already have an explicit binary representation
  - Use “U” (or “u”) suffix to explicitly force *unsigned*
    - **Examples:** 0U, 4294967259u

- Expression Evaluation
  - When you mixed unsigned and signed in a single expression, **signed values are implicitly cast to unsigned**
  - Including comparison operators <, >, ==, <=, >=

Careful!!

- Examples for 32-bit: **INT_MIN = -2147483648, INT_MAX = 2147483647**

Left Constant	Order	Right Constant	Interpretation
0 0000 0000 0000 0000 0000 0000 0000 0000	==	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	unsigned
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0** 0000 0000 0000 0000 0000 0000 0000 0000	signed
-1 1111 1111 1111 1111 1111 1111 1111 1111	>	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	unsigned
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	>	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	signed
2147483647U 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	unsigned
-1 1111 1111 1111 1111 1111 1111 1111 1111	>	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	signed
(unsigned) -1 1111 1111 1111 1111 1111 1111 1111 1111	>	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	unsigned
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	unsigned
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	>	**(int) 2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	signed

Careful!! When using unsigned variables!

- **Watch out for implicit casts!!**

  - Easy to make mistakes

```
for (unsigned i = cnt-2; i >= 0; i--)
  a[i] += a[i+1];
```

  - Can be very subtle (more on this define thing later)

```
for (int i = 100; i-sizeof(int)>= 0; i-= sizeof(int))
  . . .
```

1. `x < 0`	⇒	`((x*2) < 0)`	✗
2. `ux >= 0`			✓
3. `ux > -1`			✗
4. `x > y`	⇒	`-x < -y`	✗
5. `x * x >= 0`			✗
6. `x > 0 && y > 0`	⇒	`x + y > 0`	✗
7. `x >= 0`	⇒	`-x <= 0`	✓
8. `x <= 0`	⇒	`-x >= 0`	✗

Try to answer the following questions

Q: What would happen if you used `"%d"` for both. What is the result?

```
int main() {
  printf("%d ",  INT_MAX);
  printf("%u\n", UINT_MAX);
  return 0;
} // Output: 2147483647 4294967295
```

Q: What is the range of a `signed char`?

Q: What is the range of an `unsigned int`?

Q: Is

-3 1111 1111 1111 1111 1111 1111 1111 1101	>	**(int) 2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000

With the person next to you:
Take 2 minute to think about these questions.

I'm going to ask someone to explain

Floating point in C

And why you should avoid it :)

Floating Point in C

- C Guarantees Two Levels
  - **`float`** single precision
  - **`double`** double precision
- Conversions/Casting
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representation
  - **`double`**/**`float`** → **`int`**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - **`int`** → **`double`**
    - Exact conversion, as long as **`int`** has ≤ 53 bit word size
  - **`int`** → **`float`**
    - Will round according to rounding mode

Floating Point and the Programmer

```c
#include <stdio.h>

int main()
{
  float f1 = 1000000000.0;
  float f2 = 1.0;

  printf("f1 = %10.9f\n", f1);
  printf("f2 = %10.9f\n\n", f2);
  printf("f1+f2-f1 = %10.9f\n", f1+f2-f1);
  printf("f1-f1+f2 = %10.9f\n", f1-f1+f2);

  return 0;
}
```

```
$ ./a.out
f1 = 1000000000.000000000
f2 = 1.000000000

f1+f2-f1 = 0.000000000
f1-f1+f2 = 1.000000000
```

Floating Point and the Programmer

```
#include <stdio.h>
int main(int argc, char* argv[]) {
  float f1 = 1.0;
  float f2 = 0.0;
  for (int i = 0; i < 10; i++)
    f2 += 1.0/10.0;
  printf("0x%08x 0x%08x\n", *(int*)&f1, *(int*)&f2);

  printf("f1 = %10.9f\n", f1);
  printf("f2 = %10.9f\n\n", f2);
  f1 = 1E30;\n f2 = 1E-30;
  float f3 = f1 + f2;
  printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );
  return 0;
}
```

```
$ ./a.out
0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119

f1 == f3? yes
```

Floating Point Summary

- Systems don't usually use floats! :whew:
- Floats also suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow
  - "Gaps" produced in representable numbers means we can lose precision, unlike `int`s
    - 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 exact equality!
- Careful when converting between `int`s and `float`s!

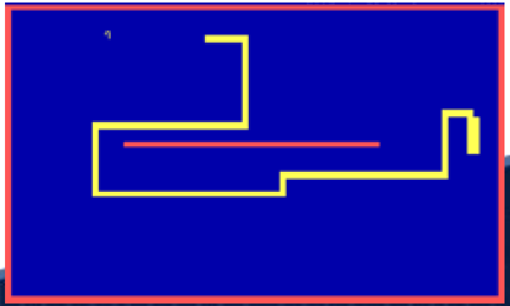
More readings

- In website:
  - https://floating-point-gui.de/basic/
  - Games are great, because they give us visual/audio feedback of these problems: https://youtu.be/PpfpfDNFdvQ?t=20
- And… you guessed it… More bonus slides :)

Bonus Slides

Positional Numbers

Bits, Bytes, and N~~y~~ibbles



- The numbers we use are written **positionally**: the position of a digit within the number has a meaning.

$$
2\,0\,2\,3 = \begin{array}{r} 2\,0\,0\,0 \\ 0\,0\,0 \\ 2\,0 \\ +\quad 3 \end{array} = \begin{array}{l} 2 \times 10^3 \\ 0 \times 10^2 \\ 2 \times 10^1 \\ 3 \times 10^0 \end{array}
$$

Positional number systems

- The numbers we use are written **positionally**: the position of a digit within the number has a meaning.

Most Significant   Least Significant

2 0 2 3

1000s	100s	10s	1s
10^3	10^2	10^1	10^0

- How many (digits) **symbols** do we have in our number system?
  - 10: 0, 1, 2, 3, 4, 5 ,6 ,7, 8, 9

A base-10 number system

Using base **10**

- A number represented by the digits

$$d_{n-1} \ldots d_1 d_0$$

- Has the value

$$d_{n-1} \times \mathbf{10}^{n-1} + \cdots + d_1 \times \mathbf{10}^1 + d_0 \times \mathbf{10}^0$$

- Using n digits we can represent $\mathbf{10}^n$ different numbers
- The smallest non-negative number representable with n digits is 0
- The largest number representable with n digits is $\mathbf{10}^n - 1$
- Using **10** symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Binary – Base 2

A quick review

Let's make a base-2 number system

Using base **2**

- A number represented by the digits

$$d_{n-1} \dots d_1 d_0$$

- Has the value

$$d_{n-1} \times \mathbf{2}^{n-1} + \cdots + d_1 \times \mathbf{2}^1 + d_0 \times \mathbf{2}^0$$

- Using n digits we can represent $\mathbf{2}^n$ different numbers
- The smallest non-negative number representable with n digits is 0
- The largest number representable with n digits is $\mathbf{2}^n - 1$
- Using **2** symbols: 0, 1

Binary (base-2)

- We call a **B**inary dig**IT** a **bit** – a single 1 or 0
- When we say an *n*-bit number, we mean one with *n* binary digits

MSB LSB

1001 0110 =

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128s	64s	32s	16s	8s	4s	2s	1s

It's the only odd number!

To convert binary to decimal: ignore 0s, add up place values wherever you see a 1.

$1 \times 128 +$
$0 \times 64 +$
$0 \times 32 +$
$1 \times 16 +$
$0 \times 8 +$
$1 \times 4 +$
$1 \times 2 +$
0×1

$= 150_{10}$

Converting the other way around

- Ok! then. Let's go back to decimal for a **bit**

2 0 2 3

How would you extract this number???

How I like to think of it:

When you divide by the BASE you are moving the decimal point in that BASE

Just divide by 10!

	0	R 2
10	2	R 0
10	2 0	R 2
10	2 0 2	R 3
10	2 0 2 3	

What happens when you divide by 10?

- Turns out that dividing by 10 in any base has the same outcome

$$0b10 \longrightarrow 2_{10}$$

	0	R 1
2_{10}	1	R 1
2_{10}	3	R 0
2_{10}	6	R 0
2_{10}	12	R 1
2_{10}	25	

$$11001_2$$

Bits, bytes, nibbles, and words

- A ***bit*** is one binary digit, and its unit is **lowercase** b.

- A ***byte*** is an 8-bit value, and its unit is **UPPERCASE** B.
  - This is (partially) why your 30 mega*bit* (Mbps) internet connection can only give you at most 3.57 mega*bytes* (MB) per second!

- A ***nibble*** (also nybble) is 4 bits – half of a byte
  - Corresponds nicely to a single hex digit.

- A ***word*** is the "most comfortable size" of number for a CPU.
- When we say "32-bit CPU," we mean its ***word*** size is 32 bits.
  - This means it can, for example, add two 32-bit numbers at once.

- **BUT WATCH OUT:**
  - Some things (Windows, x86) use **word** to mean **16 bits** and **double word** (or **dword**) to mean **32 bits.**

Everything in a computer is a number

- So, everything on a computer is represented in binary.
  - **everything.**

  01100101 01110110 01100101 01110010 01111001 01110100 01101000 01101001 01101110 01100111 00001010 00000000

- Java strings are encoded using **UTF-16**
  - Most letters and numbers in the English alphabet are < 128.
  - “Strings are numbers”
    - **83 116 114 105 110 103 115 32 97 114 101 32 110 117 109 98 101 114 115 0**

- **ASCII** is also pretty common (the best kind of common)
  - That’s what we will be using → 8 bit numbers represent characters
  - Letters and numbers (and most/all ascii characters) have the same value as UTF-16

Do try this at home: what does this mean?

- 71 111 111 100 32 74 111 98 0

Hexadecimal – Base 16

Binary shorthand

Shortcomings of binary and decimal

- Binary numbers can get really long, really quickly.
  - **$3{,}927{,}664_{10}$ = $11\ 1011\ 1110\ 1110\ 0111\ 0000_2$**
- But nice "round" numbers in binary look arbitrary in decimal.
  - **1000000000000000_2 = $32{,}768_{10}$**
- This is because 10 is not a power of 2!
- We could use base-4, base-8, base-16, base-32, etc.
  - Base-4 is not much concise than binary
    - e.g. **$3{,}927{,}664_{10}$ = $120\ 3331\ 2323\ 0000_4$**
  - Base-32 and up? would require 32+ symbols. Nope.
    - Well at least for humans… They are actually used!
  - **Base-8** and **base-16** look promising!

Let's make a base-~~2~~ 16 number system

Using base **16**

- A number represented by the digits

$$d_{n-1} \ldots d_1 d_0$$

- Has the value

$$d_{n-1} \times \mathbf{16}^{n-1} + \cdots + d_1 \times \mathbf{16}^1 + d_0 \times \mathbf{16}^0$$

- Using n digits we can represent $\mathbf{16}^n$ different numbers

- The smallest non-negative number representable with n digits is 0

- The largest number representable with n digits is $\mathbf{16}^n - 1$

- Using **16** symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Hexadecimal, or "hex" (base-16)

- Digit symbols after 9 are A-F, meaning 10-15 respectively.
- Usually we call one hexadecimal digit a *hex digit*. No fancy name :(

$$\underset{16^7}{0}\;\underset{16^6}{0}\;\underset{16^5}{3}\;\underset{16^4}{B}\;\;\underset{16^3}{E}\;\underset{16^2}{E}\;\underset{16^1}{7}\;\underset{16^0}{0} =$$

$$\begin{aligned}
0 \times 16^7 &+ \\
0 \times 16^6 &+ \\
3 \times 16^5 &+ \\
11 \times 16^4 &+ \\
14 \times 16^3 &+ \\
14 \times 16^2 &+ \\
7 \times 16^1 &+ \\
0 \times 16^0 &=
\end{aligned}$$

$$3{,}927{,}664_{10}$$

BUT WE REALLY WANT TO KNOW!!

- Turns out that dividing by 10 in any base has the same outcome

$0x10 \rightarrow 16_{10}$

We can skip the division when the result < divisor

	3	R 11
16_{10}	59	R 14
16_{10}	958	R 14
16_{10}	15 342	R 7
16_{10}	245 479	R 0
16_{10}	3 927 664	

$003B\ EE70_{16}$

Convert in groups of 4 bits from binary <-> hex

0010	0100	0110	1111	1000	0110	1001	0101
2	4	6	F	8	6	9	5

$246F8695_{16}$

0x246F8695

Why?

1111 | 1111

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

$$1\times2^3+1\times2^2+1\times2^1+1\times2^0 = 8+4+2+1 = 15$$

$$(1\times2^3+1\times2^2+1\times2^1+1\times2^0)\times2^4+15$$

Factoring

$2^4=16$

$$15\times16^1+15\times16^0$$

F | F

This works with any base that is a power of 2

E.g. Base 4=2²
Split into groups of 2 bits

Floating-point number representation

Seven-five-four!

Move the point

- What if we could **float** the point around?
  - Enter scientific notation: The number **`-0.0039`** can be represented:

$$-0.39 \times 10^{-2}$$

$$-3.9 \times 10^{-3}$$

- These both represent the **same number**, but we need to move the decimal point according to the power of ten represented.

- The bottom example is in **normalized scientific notation.**
  - There is only ***one non-zero*** digit to the left of the point.

- Because the decimal point can be moved, we call this representation:

Floating point

IEEE 754

- Established in 1985, updated as recently as 2008.
- Standard for floating-point representation and arithmetic that **virtually every CPU** now uses.
- Floating-point representation is based around **scientific notation:**

$$
\begin{aligned}
1348 &= +1.348 \times 10^{+3} \\
-0.0039 &= -3.9 \times 10^{-3} \\
-1440000 &= -\underbrace{1.44}_{\text{significand}} \times 10^{+6}
\end{aligned}
$$

sign significand exponent

Binary Scientific Notation

- Scientific notation works equally well in any other base!
  - (below uses base-10 exponents for clarity)

```
          +1001 0101 = +1.001 0101 × 2^+7
          -0.001 010 = -1.010      × 2^-3
-1001 0000 0000 0000 = -1.001      × 2^+15
```

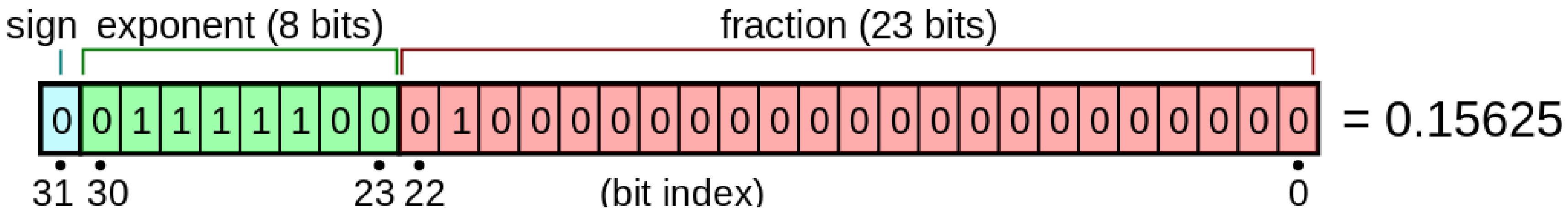
What do you notice
about the digit before
the **binary** point?

$(+/-)1.f \times 2^{exp}$

f – fraction
1.f – significand
exp – exponent

IEEE 754 Single-precision

- Known as **float** in C/C++/Java etc., 32-bit float format
- 1 bit for sign, 8 bits for the exponent, 23 bits for the *fraction*

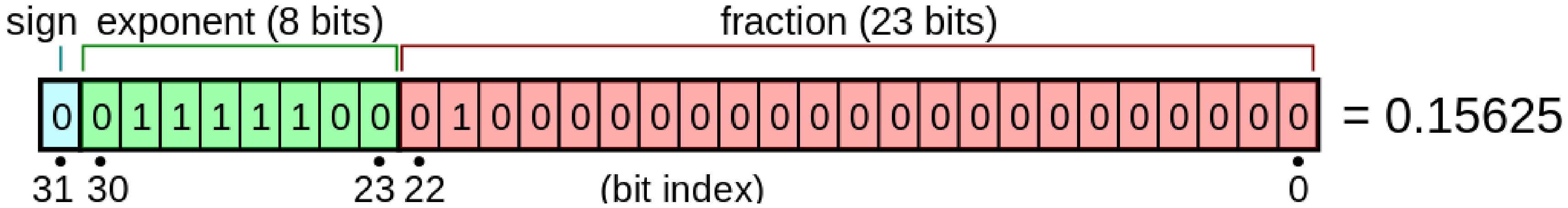


- **Tradeoff:**
  - More accuracy = More fraction bits
  - More range = More exponent bits
- **Every design has tradeoffs ¯_(ツ)_/¯**
  - Welcome to Systems!

illustration from user Stannered on Wikimedia Commons

IEEE 754 Single-precision

- Known as **float** in C/C++/Java etc., 32-bit float format
- 1 bit for sign, 8 bits for the exponent, 23 bits for the *fraction*



- **The fraction field only stores the digits after the binary point**
- **The 1 before the binary point is implicit!**
  - This is called normalized representation
  - In effect this gives us a 24-bit significand
  - The only number with a 0 before the binary point is 0!
- **The significand of floating-point numbers is in sign-magnitude!**
  - Do you remember the downside(s)?

illustration from user Stannered on Wikimedia Commons

The exponent field

- the exponent field is 8 bits, and can hold positive or negative exponents, but... it doesn't use S-M, 1's, or 2's complement.
- it uses something called **biased notation.**
  - biased representation = exponent + *bias constant*
  - **single-precision** floats use a bias constant of **127**

```
-127 + 127 => 0
exp + 127 => Biased     -10 + 127 => 117
                         34 + 127 => 161
```

- the exponent can range from **-126 to +127** (1 to 254 biased)
  - 0 and 255 are reserved!
- why'd they do this?
  - You can sort floats with integer comparisons!

Binary Scientific Notation (revisited)

- Our previous numbers are actually

$$+1.001\ 0101 \times 2^{+7} = (-1)^0 \times 1.001\ 0101 \times 2^{134-127}$$

$$-1.010 \times 2^{-3} = (-1)^1 \times 1.010 \times 2^{124-127}$$

$$-1.001 \times 2^{+15} = (-1)^1 \times 1.001 \times 2^{142-127}$$

$$(-1)^s \times 1.f \times 2^{exp-127}$$

s – sign
f – fraction
exp – biased exponent

Binary Scientific Notation (revisited)

bias = 127

$+1.001\ 0101 \times 2^{+7}$

sign = 0 (positive number!)

Biased exponent = exp + 127 = 7 + 127 = 134

= 10000110

fraction = 0010101 (ignore the "1.")

s	E	f
0	10000110	00101010000000000…000

$(-1)^0 \times 1.001\ 0101 \times 2^{134-127}$

Binary Scientific Notation (revisited)

bias = 127

$-1.010 \times 2^{-3} =$

sign = 1 (negative number!)

Biased exponent = exp + 127 = -3 + 127 = 124

= 01111100

fraction = 010 (ignore the “1.”)

s	E	f
1	01111100	01000000000000000000…000

$(-1)^1 \times 1.010 \times 2^{124-127}$

Encoding an integer as a float

- You have an integer, like 2471 = $\mathtt{0000\ 1001\ 1010\ 0111}_2$

1. put it in scientific notation
   - $\mathtt{1.001\ 1010\ 0111}_2 \times 2^{+11}$
2. get the exponent field by adding the bias constant
   - $11 + 127 = 138 = \mathtt{10001010}_2$
3. copy the bits **after the binary point** into the fraction field

s	exponent	fraction
0	10001010	00110100111000000…000

positive

start at the **left** side!

Encoding a number as a float

You have a number, like -12.59375_{10}

1. Convert to binary: Integer part: 1100_2 Fractional part: 0.10011_2

2. Write it in scientific notation: $1100.10011_2 \times 2^0$

3. Normalize it: $1.10010011_2 \times 2^3$

4. Calculate biased exponent $+3 + 127 = 130_{10} = 10000010_2$

s	exponent	fraction
1	10000010	10010011000000000…000

0xC1498000

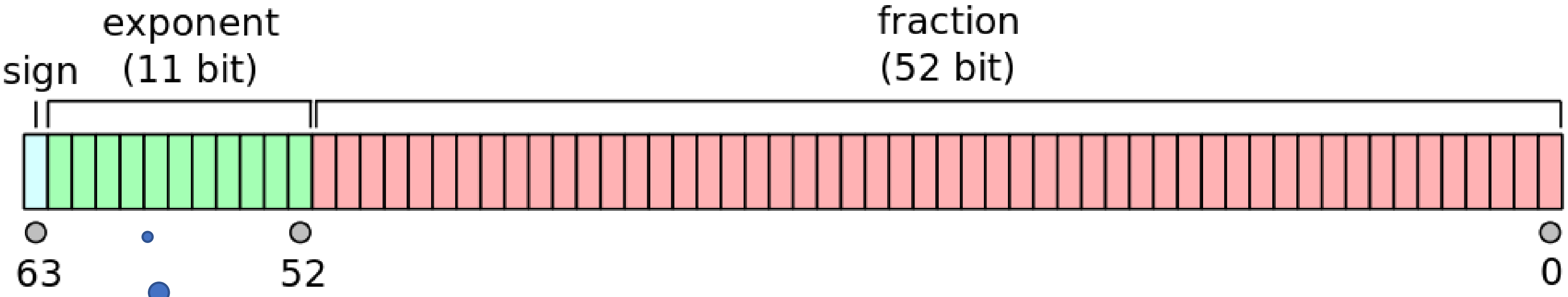
while (computers don’t do real math) { … }

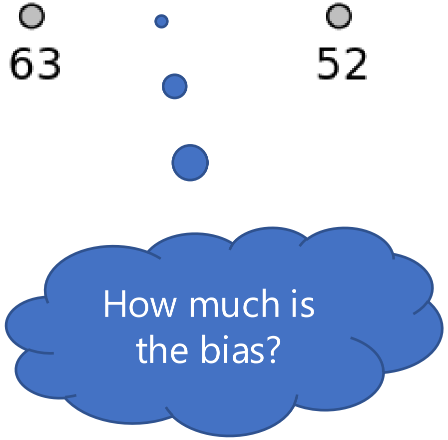
```java
public class FloatTest {
  public static void main(String[] args) {
    double var = Double.MAX_VALUE;

    while (var == Double.MAX_VALUE) {
      // Hang the program while the condition holds
      var = var + 0.1;
    }

    System.out.println("This never prints.");
  }
}
```

Q: Consider and/or review the IEEE 754 standard. What is happening here?

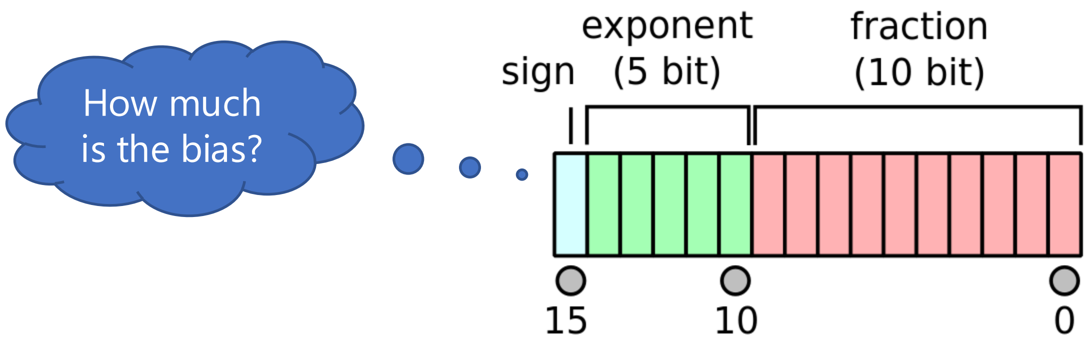
Other formats

- The most common other format is **double-precision** (C/C++/Java **double**), which uses an 11-bit exponent and 52-bit fraction





- **GPUs have driven the creation of a half-precision 16-bit floating-point format. it's adorable**



both illustrations from user Codekaizen on Wikimedia Commons

Special cases

- IEEE 754 can represent data outside of the norm.
  - Zero! How do you do that with normalized numbers?
  - +/- Infinity
  - NaN (Not a number). E.g. when you divide zero by zero.
  - Other denormalized number: Squeeze the most out of our bits!
    - E.g.: $0.0000000000000000000000001 \times 2^{-126}$

Single precision		Double precision		Meaning
Exponent	**Fraction**	**Exponent**	**Fraction**	
0	0	0	0	0
0	!=0	0	!=0	Number is denormalized (The exponent is -126/1022)!
255	0	2047	0	Infinity (sign-bit defines + or -)
255	!=0	2047	!=0	NaN (Not a Number)

This could be a whole unit itself...

- Floating-point arithmetic is COMPLEX STUFF.

- But it's not super useful to know unless you're either:
  - Doing lots of high-precision numerical programming, or
  - Implementing floating-point arithmetic yourself.

- However...
  - It's good to have an understanding of *why* limitations exist.
  - It's good to have an *appreciation* of how complex this is... and how much better things are now than they were in the 1970s and 1980s!
  - It’s good to know things do not behave as expected when using float and double!!

Q: Which is IEEE754 double precision number immediately before +4.0 ?

Interesting Numbers

Description	*exp*	*frac*	*Numeric Value*
• Zero	00…00	00…00	0.0
• Smallest Pos. Denorm.	00…00	00…01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
• Single ≈ 1.4×10^{-45}			
• Double ≈ 4.9×10^{-324}			
• Largest Denormalized	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
• Single ≈ 1.18×10^{-38}			
• Double ≈ 2.2×10^{-308}			
• Smallest Pos. Normalized	00…01	00…00	$1.0 \times 2^{-\{126,1022\}}$
• Just larger than largest denormalized			
• One	01…11	00…00	1.0
• Largest Normalized	11…10	11…11	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$
• Single ≈ 3.4×10^{38}			
• Double ≈ 1.8×10^{308}			

`{single,double}`

Dynamic Range (s=0 only)

$v = (-1)^s M 2^E$
norm: $E = exp - Bias$
denorm: $E = 1 - Bias$

	s exp frac	E	Value	
Denormalized numbers	0 0000 000	-6	0	
	0 0000 001	-6	1/8*1/64 = 1/512	closest to zero
	0 0000 010	-6	2/8*1/64 = 2/512	$(-1)^0(0+1/4)*2^{-6}$
	…			
	0 0000 110	-6	6/8*1/64 = 6/512	
	0 0000 111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0 0001 000	-6	8/8*1/64 = 8/512	smallest norm
	0 0001 001	-6	9/8*1/64 = 9/512	$(-1)^0(1+1/8)*2^{-6}$
	…			
	0 0110 110	-1	14/8*1/2 = 14/16	
	0 0110 111	-1	15/8*1/2 = 15/16	closest to 1 below
	0 0111 000	0	8/8*1 = 1	
	0 0111 001	0	9/8*1 = 9/8	closest to 1 above
	0 0111 010	0	10/8*1 = 10/8	
	…			
	0 1110 110	7	14/8*128 = 224	
	0 1110 111	7	15/8*128 = 240	largest norm
	0 1111 000	n/a	inf	

Rounding

1.BBGRXXX

Guard bit: LSB of result

Round bit: 1st bit removed

Sticky bit: OR of remaining bits

- **Round up conditions**
  - Round = 1, Sticky = 1 → > 0.5
  - Guard = 1, Round = 1, Sticky = 0 → Round to even

Fractio	***GRS***	***Incr?***	***Rounded***
1.0000000	000	N	1.000
1.1010000	100	N	1.101
1.0001000	010	N	1.000
1.0011000	110	Y	1.010
1.0001010	011	Y	1.001
1.1111100	111	Y	10.000