15-213 Recitation: Data Lab

Jack Biggs

25 Jan 2016

Agenda

- Introduction
- Course Details
- Data Lab
  - Getting started
  - Running your code
  - ANSI C
- Bits & Bytes
- Integers
- Floating Point

Introduction

- Welcome to 15-213/18-213/15-513!
- Recitations are for…
  - Reviewing lectures
  - Discussing homework problems
  - Interactively exploring concepts
  - Previewing future lecture material

- Please, **please** ask questions!

Course Details

- How do I get help?
  - Course website: http://cs.cmu.edu/~213
  - Office hours: **5-9PM** from Sun-Thu in Wean 5207
  - Staff mailing list: 15-213-staff@cs.cmu.edu
  - *Definitely* consult the course textbook
  - **Carefully read the assignment writeups!**
- All labs are submitted on Autolab.
- All labs should be worked on using the **shark machines.**

Data Lab: Getting Started

- Download lab file (`datalab-handout.tar`)
  - Upload tar file to **shark** machine
  - `cd <my course directory>`
  - `tar xpvf datalab-handout.tar`
- `<filename>: Permission denied`
  - `chmod +x <filename>`
- Upload `bits.c` file to Autolab for submission

Data Lab: Running your code

- `dlc`: a modified C compiler that interprets *ANSI C* **only**
- `btest`: runs your solutions on random values
- `bddcheck`: exhaustively tests your solutions
  - Checks all values, formally verifying the solution
- `driver.pl`: Runs both dlc and bddcheck
  - Exactly matches Autolab's grading script
  - You will likely only need to submit once
- For more information, **read the writeup**
  - Available under assignment page as "**View writeup**"
  - **Read it. Read the writeup... please.**

Data Lab: What is ANSI C?

This is *not* ANSI C.

```
unsigned int foo(unsigned int x)
{
        x = x * 2;
    int y = 5;

        if (x > 5) {
        x = x * 3;
                int z = 4;
                x = x * z;
        }

        return x * y;
}
```

Within two braces, all *declarations* must go before any *expressions*.

Data Lab: What is ANSI C?

This is ANSI C.

```
unsigned int foo(unsigned int x)
{
        int y = 5;
        x = x * 2;

        if (x > 5) {
                int z = 4;
                x = x * 3;
                x = x * z;
        }

        return x * y;
}
```

This is *not* ANSI C.

```
unsigned int foo(unsigned int x)
{
        x = x * 2;
    int y = 5;

        if (x > 5) {
       x = x * 3;
               int z = 4;
               x = x * z;
        }

        return x * y;
}
```

Bits & Bytes: Unsigned integers

- An unsigned number represents positive numbers between 0 and 2^k-1, where *k* is the numbers of bits used.
- Subtracting 1 from 0 will *underflow* to the highest value.
- Adding 1 to the highest value will *overflow* to 0

An 8-bit unsigned integer:

1	0	0	1	0	1	0	1

$2^7 + 2^4 + 2^2 + 2^0 = 149$

Bits & Bytes: Two's Complement

- In C, a *signed* number represents numbers between [-2^{k-1}, $2^{k-1}-1$], where *k* is the number of bits used.
- Overflow and underflow (from max > 0 and min < 0 values) is *undefined* with signed numbers in C
  - Depending on the underlying architecture, signed overflow / underflow could modulo, do nothing, or even abort the program
- The highest-level bit is set to 1 in negative numbers.
- To get the negative value of a positive number *x*, invert the bits of *x* and add 1.

Bits & Bytes: Two's Complement

From positive to negative:

0	**0**	**0**	**1**	**0**	**1**	**0**	**1**

Bits & Bytes: Two’s Complement

From positive to negative:

0	**0**	**0**	**1**	**0**	**1**	**0**	**1**	
1	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**Bits negated**

Bits & Bytes: Two’s Complement

From positive to negative:

0	0	0	1	0	1	0	1	
1	1	1	0	1	0	1	0	**Bits negated**
1	1	1	0	1	0	1	1	**Add one**

Bits & Bytes: Two's Complement

From negative to positive:

1	**1**	**0**	**1**	**0**	**1**	**0**	**1**

Bits & Bytes: Two’s Complement

From negative to positive:

1	1	0	1	0	1	0	1	
0	0	1	0	1	0	1	0	**Bits negated**

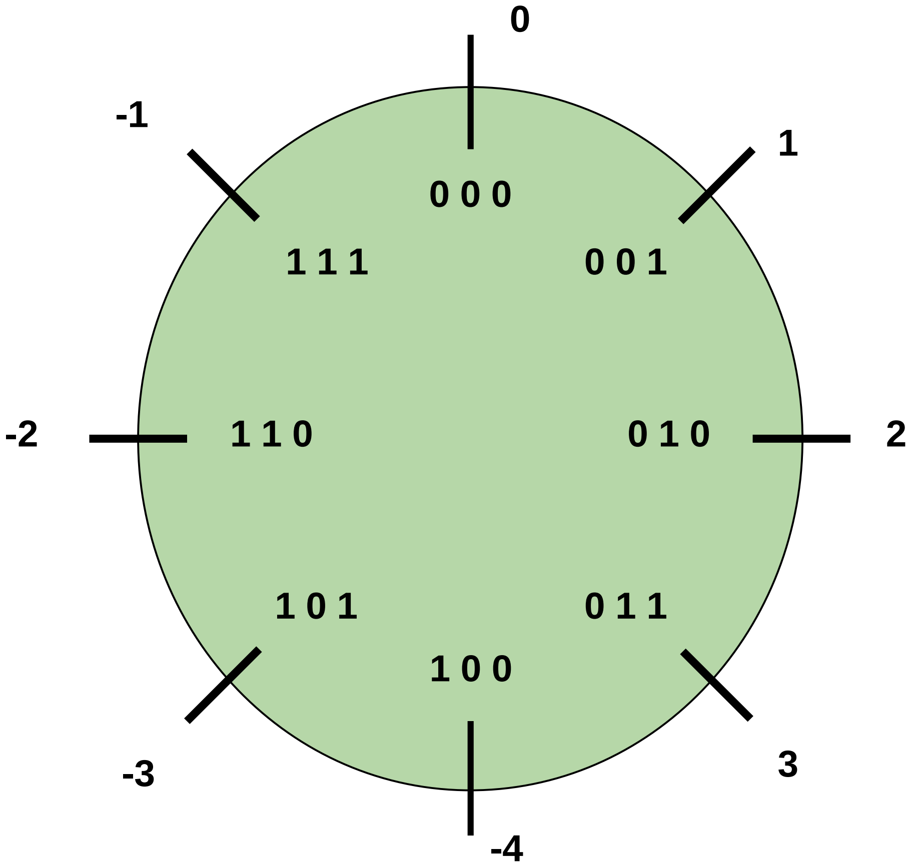
Bits & Bytes: Two's Complement

From negative to positive:

1	1	0	1	0	1	0	1	
0	0	1	0	1	0	1	0	**Bits negated**
0	0	1	0	1	0	1	1	**Add one**

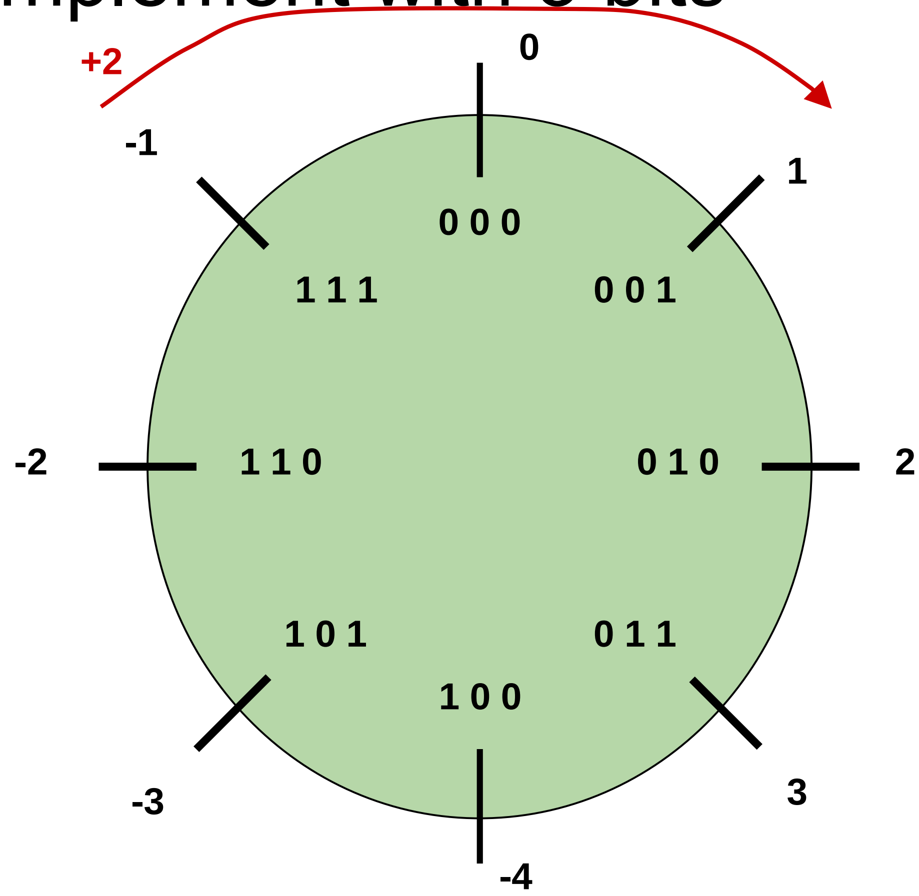
Bits & Bytes: Two's Complement with 3 bits

- Why would anybody want to do this?
  - Uses the same circuitry for addition and subtraction!
- Note that there is no positive 4: the two's complement of -4 with three bits is -4



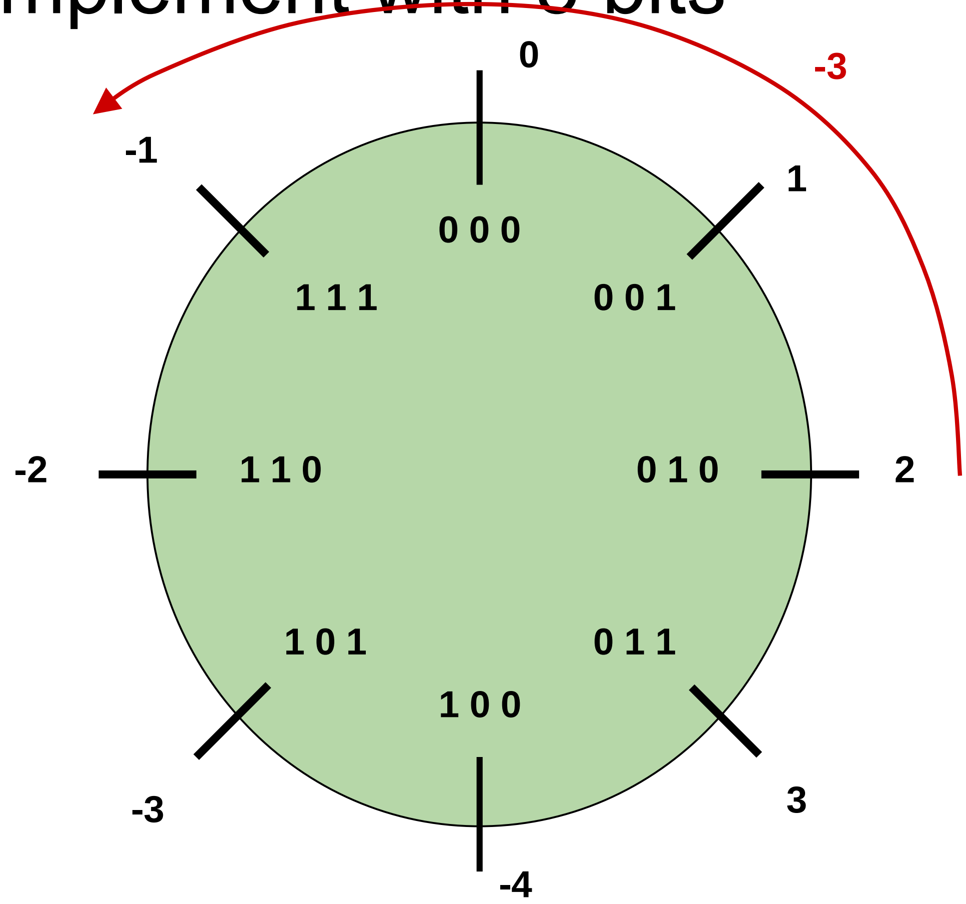
Bits & Bytes: Two's Complement with 3 bits

- Why would anybody want to do this?
  - **Uses the same circuitry for *addition* and subtraction!**
- Note that there is no positive 4: the two's complement of -4 with three bits is -4



Bits & Bytes: Two's Complement with 3 bits

- Why would anybody want to do this?
  - **Uses the same circuitry for addition and *subtraction*!**
- Note that there is no positive 4: the two's complement of -4 with three bits is -4



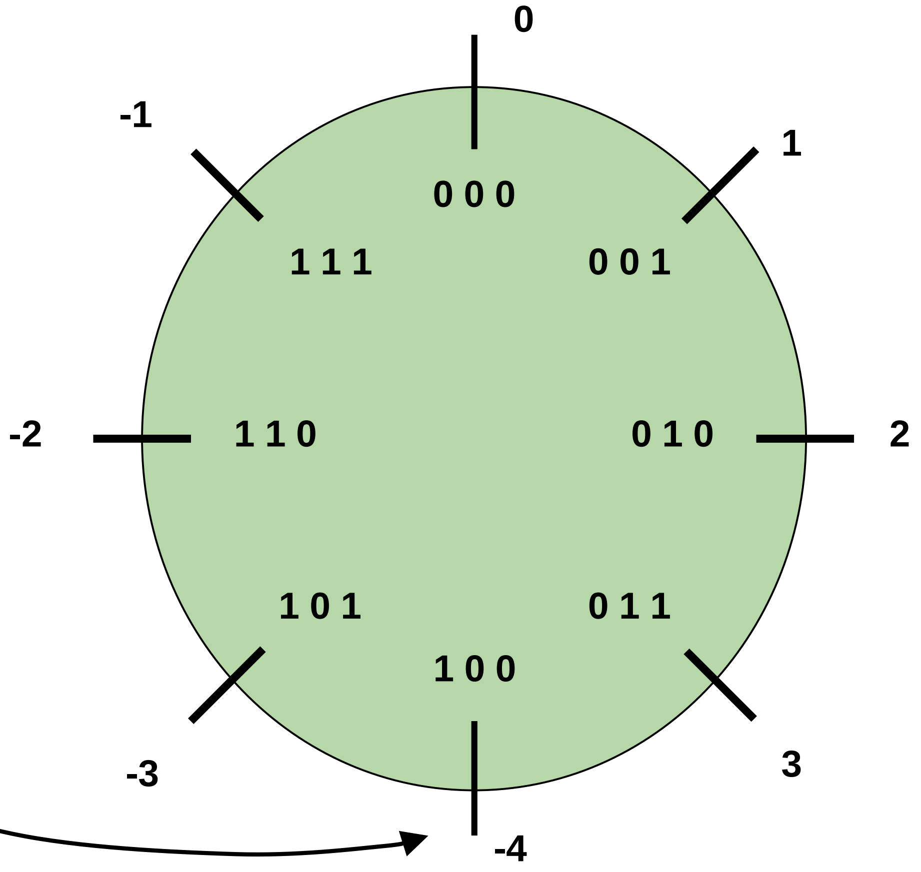
Bits & Bytes: Two's Complement with 3 bits

<u>Negating The Minimum</u>

-4 = 100

~(-4) = 011

~(-4) + 1 = 100



Bits & Bytes: Logical Operators

AND: &&	OR: ||	EQ: ==	NOT: !
15 && 18 =	513 || 0 =	15 == 18 =	!15213 =

Bits & Bytes: Logical Operators

AND: &&	OR: ||	EQ: ==	NOT: !
15 && 18 = 1	513 || 0 =	15 == 18 =	!15213 =

Bits & Bytes: Logical Operators

AND: &&	OR: ||	EQ: ==	NOT: !
15 && 18 = 1	513 || 0 = 1	15 == 18 =	!15213 =

Bits & Bytes: Logical Operators

AND: &&	OR: ||	EQ: ==	NOT: !
15 && 18 = 1	513 || 0 = 1	15 == 18 = 0	!15213 =

Bits & Bytes: Logical Operators

AND: &&	OR: ||	EQ: ==	NOT: !
15 && 18 = 1	513 || 0 = 1	15 == 18 = 0	!15213 = 0

Bits & Bytes: Bitwise Operators

AND: &	OR: |	XOR: ^	NOT: ~
01100101	01100101	01100101	
& 11101101	| 11101101	^ 11101101	~11101101

Bits & Bytes: Bitwise Operators

AND: &	OR: |	XOR: ^	NOT: ~
01100101	01100101	01100101	
& 11101101	| 11101101	^ 11101101	~11101101
01100101			

Bits & Bytes: Bitwise Operators

AND: &

```
  01100101
& 11101101
----------
  01100101
```

OR: |

```
  01100101
| 11101101
----------
  11101101
```

XOR: ^

```
  01100101
^ 11101101
----------
```

NOT: ~

```
~11101101
---------
```

Bits & Bytes: Bitwise Operators

AND: &	OR: |	XOR: ^	NOT: ~
01100101	01100101	01100101	
& 11101101	| 11101101	^ 11101101	~11101101
01100101	11101101	10001000	

Bits & Bytes: Bitwise Operators

AND: &	OR: |	XOR: ^	NOT: ~
01100101	01100101	01100101	
& 11101101	| 11101101	^ 11101101	~11101101
01100101	11101101	10001000	00010010

Bits & Bytes: Shifting

Shifting modifies the positions of bits in a number:

1	0	1	1	0	0	0	1	0	1	0	1	1	0	0	1	1	= x
0	0	0	1	0	1	0	1	1	0	0	1	1	0	0	0	0	= x << 4

~~1 0 1 1~~

$= x * 2^4$

Shifting right on a signed number will *extend the sign:*

x =	1	0	1	1	0	0	0	1	0	1	0	1	1	0	0	1	1
x >> 4 =	1	1	1	1	1	0	1	1	0	0	0	1	0	1	0	1	1

~~0 0 1 1~~

$x / 2^4$ =

(If the sign bit is zero, it will fill in with zeroes instead.)

This is known as “arithmetic” shifting.

Bits & Bytes: Shifting

Shifting right on an *unsigned* number will fill in with 0.

x =	1	0	1	1	0	0	0	1	0	1	0	1	1	0	0	1	1	
x >> 4 = **$x / 2^4$ =**	0	0	0	0	1	0	1	1	0	0	0	1	0	1	0	1	1	~~0 0 1 1~~

This is known as "logical" shifting.

Arithmetic shifting is useful for preserving the sign when dividing by a power of 2.

We get around this when we don't need it by using *bitmasks.*

In other languages, such as Java, it is possible to choose shifting operators, regardless of the type of integer. In C, however, it depends on the signedness.

Bits & Bytes: Endianness (Byte Order)

- Endianness describes which byte in a number comes first in memory. This is important for bomb lab.
- Little-Endian machines store the lowest-order byte first.
  - Intel machines (the shark machines!) are little-endian.

`0xdeadbeef:` `0x ef be ad de`

- Big-Endian machines store the highest-order byte first.
  - The Internet is big-endian
  - How we think about binary numbers normally

`0xdeadbeef:` `0x de ad be ef`

Floating Point: "Fixed" Point Representation

1	1	0	1	.	0	1	1	0	
2^3 4	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	2^-	
8 +	4	+	1		+	¼ +	⅛		= 13 ⅝

- Bits to the right of the "binary point" represent smaller fractions
- Difficult to represent a wide range of numbers
  - In this example, can't represent a number larger than 16
  - Can we sacrifice a bit of precision to accomplish this?

Floating Point: Scientific Notation

- In Scientific Notation, we represent a number as a fraction multiplied by an exponentiated scaling factor.

In base 10: $1.5213 * 10^7 = 15{,}213{,}000$

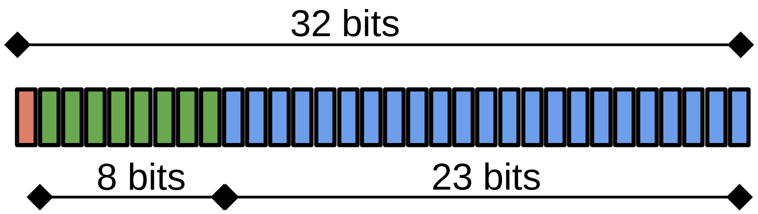
mantissa / fraction / significand
(choose what you want to call it)

exponent

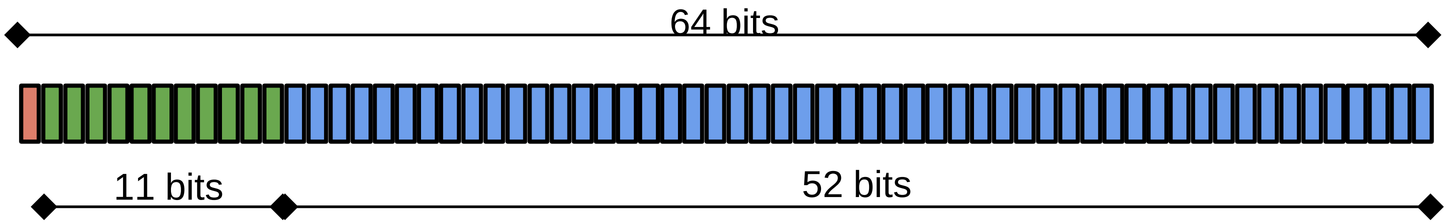
In binary: $1.011_2 * 2^{13} = (1 + \frac{1}{4} + \frac{1}{8}) * 8192 = 11264$

Floating Point: IEEE Standard

In C:



`float`



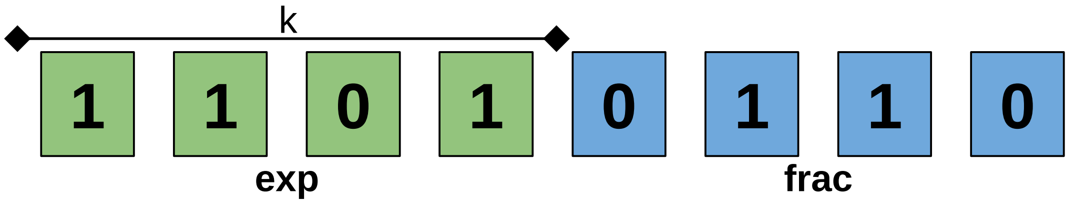
`double`

Floating Point: Sign and Exponent

- If sign is 1, then the number is negative.
- The exponent determines three different value types.
  - Normalized: `0 != exp != 1111`$_2$...
    - Mantissa = s * 1.frac
  - Denormalized: `exp = 0`
    - Mantissa = s * 0.frac
  - Special: exp = 1111_2…
- ***Neither*** exp nor frac use two’s complement!

Floating Point Example: Normalized

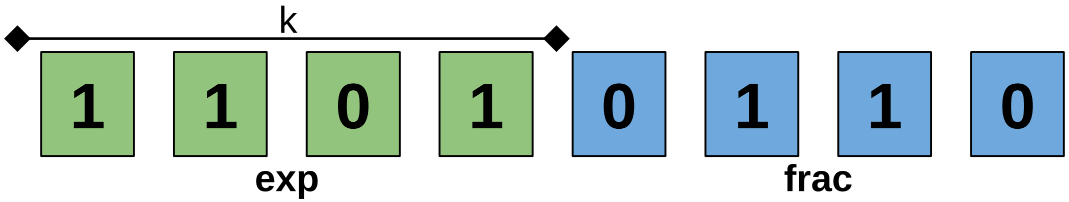
Consider a floating point implementation based on the IEEE floating point standard. This implementation omits the sign bit, and uses 4 bits for the exponent and 4 bits for the fraction.



E = exp - bias, where bias = 2^{k-1}-1.

Floating Point Example: Normalized

Consider a floating point implementation based on the IEEE floating point standard. This implementation omits the sign bit, and uses 4 bits for the exponent and 4 bits for the fraction.

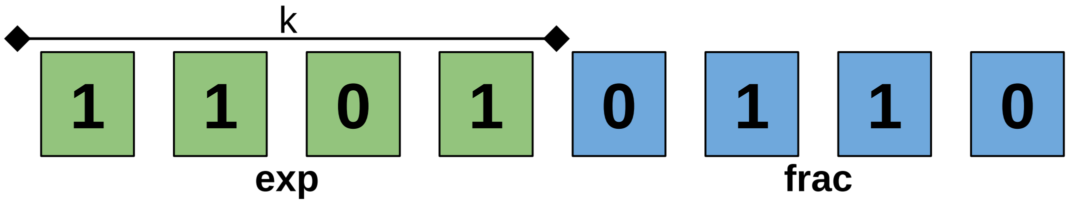


E = exp - bias, where bias = 2^{k-1}-1.

bias = 2^{4-1}-1 = 7.

Floating Point Example: Normalized

Consider a floating point implementation based on the IEEE floating point standard. This implementation omits the sign bit, and uses 4 bits for the exponent and 4 bits for the fraction.



E = exp - bias, where bias = 2^{k-1}-1.

bias = 2^{4-1}-1 = 7.

E = 13 - 7 = 6

Floating Point Example: Normalized

Consider a floating point implementation based on the IEEE floating point standard. This implementation omits the sign bit, and uses 4 bits for the exponent and 4 bits for the fraction.

k

exp				frac			
1	**1**	**0**	**1**	**0**	**1**	**1**	**0**

E = exp - bias, where bias = $2^{k-1}-1$.

bias = $2^{4-1}-1 = 7$.

E = 13 - 7 = 6

Fraction has an implied leading 1.

Floating Point Example: Normalized

Consider a floating point implementation based on the IEEE floating point standard. This implementation omits the sign bit, and uses 4 bits for the exponent and 4 bits for the fraction.

k

1	1	0	1	0	1	1	0

exp **frac**

E = exp - bias, where bias = 2^{k-1}-1.

bias = 2^{4-1}-1 = 7.

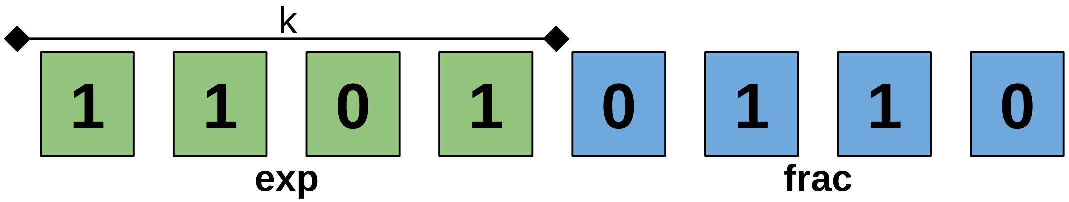
E = 13 - 7 = 6

Fraction has an implied leading 1.

Mantissa = 1.0110_2

Floating Point Example: Normalized

Consider a floating point implementation based on the IEEE floating point standard. This implementation omits the sign bit, and uses 4 bits for the exponent and 4 bits for the fraction.



E = exp - bias, where bias = $2^{k-1}-1$.

bias = $2^{4-1}-1 = 7$.

E = 13 - 7 = 6

Fraction has an implied leading 1.

Mantissa = 1.0110_2

$2^6 * 1.0110_2 = 64 * (1 + ¼ + ⅛) = \mathbf{88}$

Floating Point Example: Denormalized

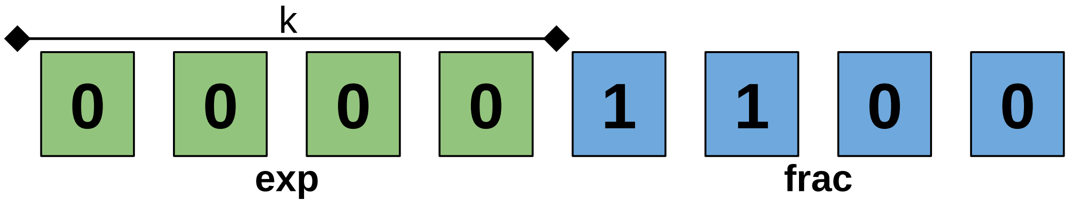
k

0	0	0	0	1	1	0	0

exp frac

E = 1 - bias, and bias = 7 as before.

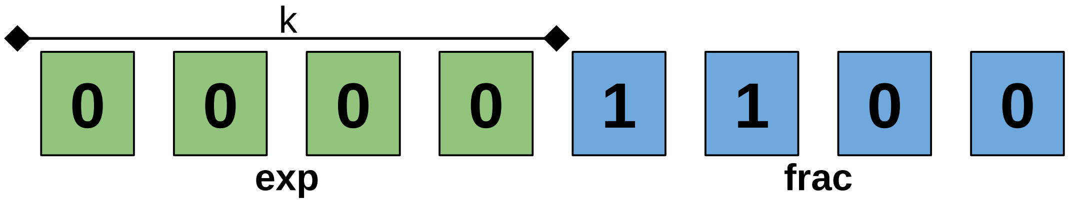
Floating Point Example: Denormalized



E = 1 - bias, and bias = 7 as before.

Fraction has an implied leading 0.

Floating Point Example: Denormalized

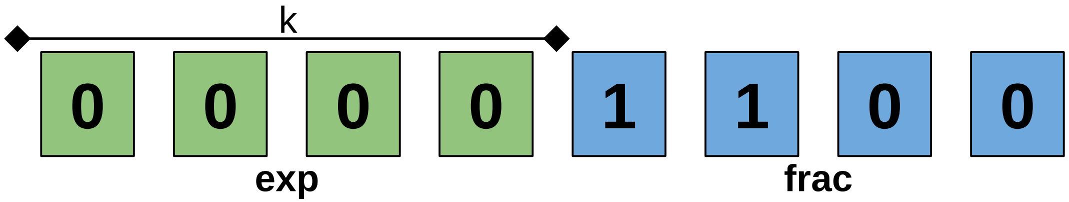


E = 1 - bias, and bias = 7 as before.

Fraction has an implied leading 0.

Fraction is equal to 0.1100_2

Floating Point Example: Denormalized



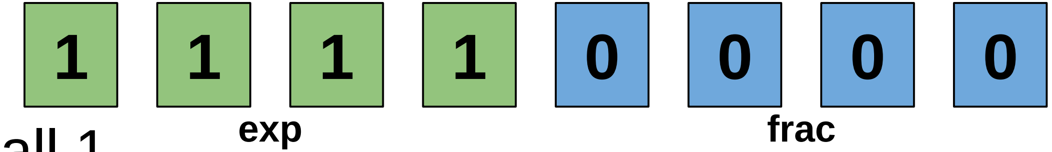
E = 1 - bias, and bias = 7 as before.

Fraction has an implied leading 0.

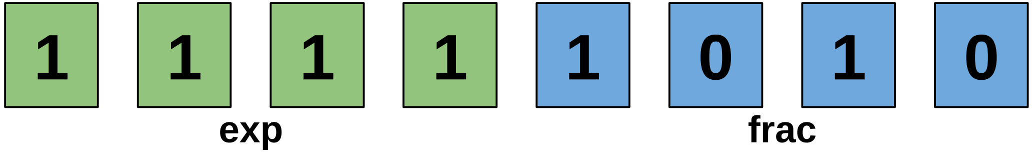
Fraction is equal to 0.1100_2

Final answer: 2^{-6} * (0 + ½ + ¼) = **0.01171875**

Floating Point Example: Special Values

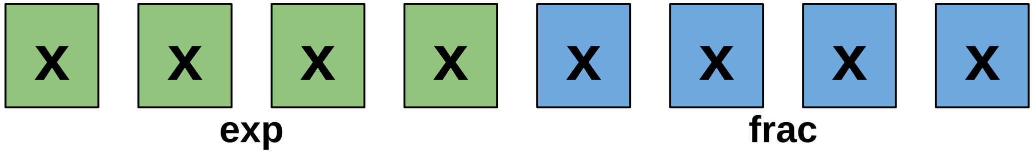


- Exp is all 1
- If fraction is all 0, then represents infinity
  - Also, -Infinity (if we had a sign bit)



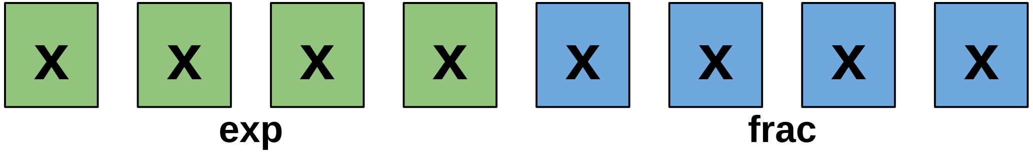
- If fraction != 0, then represents NaN (Not a Number!)
- Sign bit doesn't *really* matter, but either can turn up
  - (Mostly from division errors)

Floating Point Example: Limits



- What is the largest denormalized number?
- What is the smallest normalized number?
- What is the largest finite number it can represent?
- What is the smallest non-zero value it can represent?

Floating Point Example: Limits

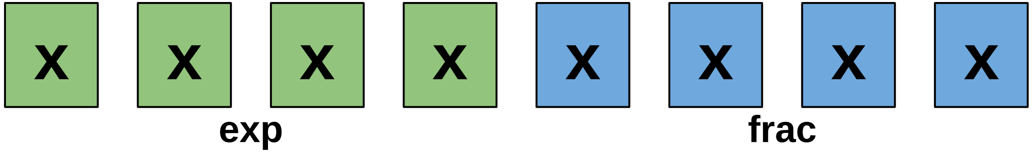


- **What is the largest denormalized number?**
- What is the smallest normalized number?
- What is the largest finite number it can represent?
- What is the smallest non-zero value it can represent?

$$\mathbf{0000\ 1111 = 0.1111_2 * 2^{-6} = 0.0146484375}$$

(recall that E = 1 - bias, and bias = 7 in this example)

Floating Point Example: Limits



- What is the largest denormalized number?
- **What is the smallest normalized number?**
- What is the largest finite number it can represent?
- What is the smallest non-zero value it can represent?

0001 0000

E = 1 - 7 = -6

Answer: $1.0000_2 * 2^{-6} = 2^{-6} = 1/64$

Floating Point Example: Limits

exp				frac			
x	x	x	x	x	x	x	x

- What is the largest denormalized number?
- What is the smallest normalized number?
- **What is the largest finite number it can represent?**
- What is the smallest non-zero value it can represent?

1110 1111

$E = 14 - 7 = 7$

Answer: $1.1111_2 * 2^7 = 248$

Floating Point Example: Limits

x	x	x	x	x	x	x	x

exp frac

- What is the largest denormalized number?
- What is the smallest normalized number?
- What is the largest finite number it can represent?
- **What is the smallest non-zero value it can represent?**

0000 0001

$$= 0.0001_2 * 2^{-6} = 0.0009765625$$

(recall that E = 1 - bias, and bias = 7 in this example)

Floating Point: Rounding

1.BBGRXXX

In the below examples, imagine the underlined part as a fraction.

- Guard Bit: the least significant bit of the resulting number
- Round Bit: the first bit removed from rounding
- Sticky Bits: all bits after the round bit, OR'd together

Examples of rounding cases, including rounding to nearest even number

- 1.10 11: More than ½, round up: 1.11
- 1.10 10: Equal to ½, round down *to even*: 1.10
- 1.01 01: Less than ½, round down: 1.01
- 1.01 10: Equal to ½, round up *to even:* 1.10
- 1.01 00: Equal to 0, do nothing: 1.01
- 1.00 00: Equal to 0, do nothing: 1.00

All other cases involve either rounding up or down - *try them*!

Questions?

- Remember, data lab is due this Thursday!
  - You really should have started already!
- Read the lab writeup.
  - **Read the lab writeup.**
    - ***Read the lab writeup.***
      - ***<u>Read the lab writeup.</u>***
        - **<u>Please. :)</u>**