note @74 stop following 133 views

Actions

DATALAB: Tips & stuff I've learned about

Hello all! Welcome to CS 356! I've really enjoyed this class, as a student and *especially* as a teacher, so I wanted to give back a little for my last semester here at USC.

This class provides a good opportunity to understand coding down to the very fine, low-level details, which I think can add a level of understanding that you can appreciate for whichever career you ambitiously pursue. Which I think is pretty cool.

So I hope that you can share some of my enthusiasm here. I've seen other students really enjoy the class too. For some, the content may not be aligned with your interests or your strengths, and that's ok. I promise that you can still do well, and there's a whole great staff here eager to help you.

Intro

Something that was really an "aha" moment for me with this assignment is that **data can be interpreted in many different ways**. Let's jump right into an example.

```
unsigned num = 0xCAFEBABE;
```

What data type is **num**?
a) int
b) unsigned int
c) float

Before we do that, though, let's break down exactly what **0xCAFEBABE** is. the **0x** prefix means that we are interpreting the following as "hexadecimal," or base-16, which is represented by digits (0-9, and A-F for 10-15). The alternatives are regular decimal, digits (0-9), or binary, digits (0-1). Let's try representing this number in our different number systems.

Decimal: You should reference the Unit 1 slides to remember how to convert between number systems, but for this example:
0xCAFEBABE = E(14) *** 16^0 + B**(11) *** 16^1 + ... + C**(12) *** 16^7 = 3405691582**
(good way to remember the hex digits: C-12, Carbon's atomic number is 12 and it's in the middle so can go up or down from there)

Binary: Remember that each hex digit is represented by 4 binary digits... since each hex digit is 0-15, 16 options, we need 4 binary digits for that. **Two hex digits make up a byte.** So we can start with the C and work our way over. We should have 8 4-digit blocks of binary.
0xCAFEBABE = 0b 1100 1010 1111 1110 1011 1010 1011 1110

If you didn't know, we can actually declare variables using binary using the **0b** prefix, so **these three declarations are all the same:**

```
unsigned num = 0xCAFEBABE;
unsigned num = 3405691582;
unsigned num = 0b11001010111111101011101010111110;
```

Ok enough stalling, let's answer the question. Well it's a trick question of course! The data can be **interpreted** in many different ways, which is really important for this assignment. For example, in the floating-point puzzles, they are declared as unsigned, but we are interpreting them as floats. So the compiler just sees it as data, but for you, me, Prof. Goodney, Prof. Paolieri, and ./grade ... we see them as floats.

It'll be the most helpful to **look at the binary for interpreting the number with respect to different data types.**

a) signed int (2's complement)
num = -889 275 714

Makes sense, the sign bit is 1 so it should be negative, it's within our range of [-2 147 483 648 to 2 147 483 647], should also be pretty big because there's ones and zeroes scattered throughout I guess, idk ¯_(ツ)_/¯.

b) unsigned int
num = 3 405 691 582.

Well we see that it's not in our signed int range, but that's ok, it is in fact in our unsigned int range of [0 to 4 294 967 295]. So that's good. Not much else to comment on here.

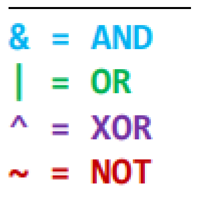
c) float
num = -8346975.0

Also makes sense here that num is negative because float is a negative data type. It also *magically* worked out that there's nothing after the decimal part because my random example *magically* has '22' in the exponent bits, and since there's 23 fraction bits, we move the decimal 22 bits over, and also the last bit of num is 0 so that's why it's xxx.0. Kind of a wild coincidence here. Bonus points if you can understand all the *magic* happening here.

Lastly, life is short. Shoot your shot with the café babe :)

Manipulating data and 'masking'

So now that we've established that we can get really *in there* with the data we're working with, all the way down to the ones and zeroes, let's talk about how to operate on the data, one bit at a time. These are known as 'bitwise operators.'



& = AND
| = OR
^ = XOR
~ = NOT

Unit 2, slides 10-11 are crucial for this assignment. Make sure you understand them.

Let's go through an example for masking.

```
0110 ???? ???? 1110 0101 1100 1111
```

Let's say I wanted to **preserve** the ???? and **force 1** for all the other bits.

What operator do I use to force 1?
--> **OR**. Notice that for ? = 0/1, **? | 1 = 1**.

What operator do I use to preserve the ???? ?
--> **Either.** Notice that for ? = 0/1, **? | 0 = ?** and **? & 1 = ?**. But since we are **creating a mask**, and we want to **force 1** for the other bits, **we need OR** because there is no way to force 1 with the AND operator.

So, let's give it a shot.

```
        0110 ???? ???? 1110 0101 1100 1111                      <--- the number we're trying t
o manipulate
|       1111 0000 0000 1111 1111 1111 1111                      <--- the mask that we create t
o do the job
-------------------------------------
        1111 ???? ???? 1111 1111 1111 1111                      <--- the result we're after
```

Here's our example, but in code. In words, here is our task: **Preserve the 3rd byte, and force the other bits to 1.**

```
// given =    0110 ???? ???? 1110 0101 1100 1111
int mask = 0b 1111 1111 0000 1111 1111 1111 1111;
return given | mask;
```

Okokok I tricked you on this one a little bit. The code has a minor mistake that causes big problems, but essentially, **the mask is only 3.5 bytes, but the compiler will always read it as 4.** This means we would actually get

```
given | mask
= 0000 1111 ???? ???? 1111 1111 1111 1111
```

Which is not what we want. I also tricked you a little bit by only giving you 3.5 bytes of the given without the leading zeroes, but I wanted to point this out. **Remember to account for all 4 bytes! You can think of the number with leading zeroes to fill up the full 4 bytes.** Here's the same example but without the deception.

```
// given =    0000 0110 ???? ???? 1110 0101 1100 1111
int mask = 0b 1111 1111 1111 0000 1111 1111 1111 1111;  //now making sure that the full 4 byte
s are exactly what we want
return given | mask;

// given | mask = 1111 1111 ???? ???? 1111 1111 1111 1111
```

Reference the slides for how each of the other operations are useful in masking.

-- AND (&) -- forcing certain bits to be zero, preserving the rest

-- OR (|) -- forcing certain bits to be one, preserving the rest

-- XOR (^) -- flipping certain bits, preserving the rest

-- NOT (~) -- flipping all bits -- only takes one argument, so we cannot specify which bits to 'preserve' and which to 'flip' like we can with XOR (can you think of why this would still be useful? hint: how would I flip all bits using XOR?)

FOR THE FLOATING POINT PUZZLES YOU CAN DECLARE BIG NUMBERS

Suppose I wanted to work with the value **-1**.

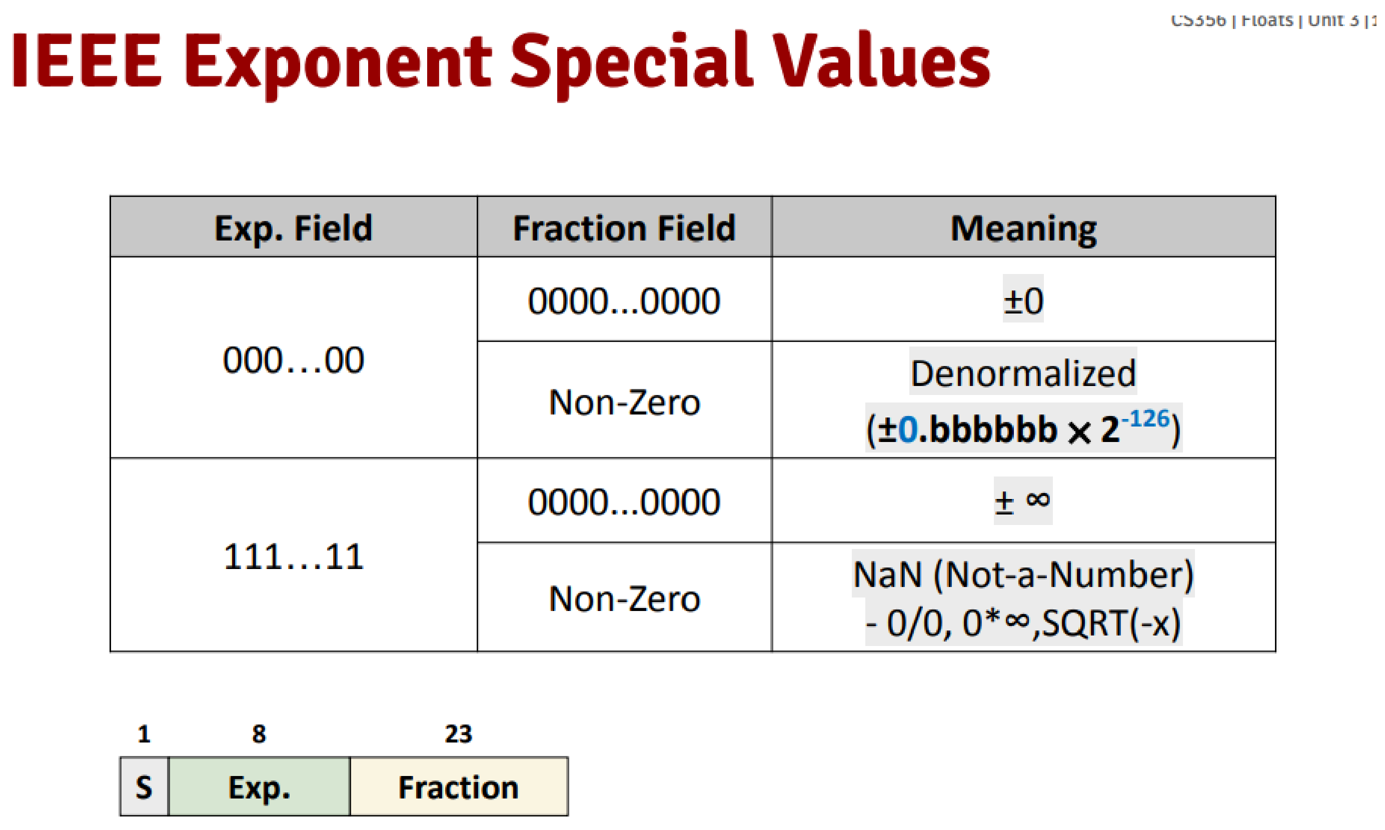
In the integer puzzles, I'd have to do this: **int negOne = ~0.**

In the floating point puzzles, I can do this: **int negOne = 0xFF FF FF FF** which saves me an operation. Useful!

Just wanted to make sure you saw this.

Denormalized / special floating point values

For some of the floating point puzzles, you'll have to work with the special floating point values, where we have to stray from the normal rules for floating point representation. I'll try to help with the understanding here.



CS356 | Floats | Unit 3 | 1

IEEE Exponent Special Values

Exp. Field	Fraction Field	Meaning
000…00	0000...0000	±0
	Non-Zero	Denormalized ($\pm 0.bbbbbb \times 2^{-126}$)
111…11	0000...0000	± ∞
	Non-Zero	NaN (Not-a-Number) - 0/0, 0*∞,SQRT(-x)

1	8	23
S	Exp.	Fraction

Unit 3 -- slide 16

So for starters, we notice that the special values are determined by the exponent bits.

Are the exponent bits all zeroes? Then it's a special value.

Are the exponent bits all ones? Then it's a special value.

Otherwise? Not a special value. Interpret the data normally.

So, we can break the special values into two categories: **too big or too small** for normal interpretation.

Too big:
What's the biggest number that we can represent normally? Well, we can't have the exponents being all the same, so the best we can do, to get the biggest exponent, is **11111110** = 254, the first 7 exponent bits being 1, and the last one being zero, which gives us our **largest exponent of 254-127 = +127**.
We're still on the mission to make the biggest number possible, so we also set all the fraction bits to be 1, which gives us our **biggest number** (magnitude-wise) to be:

S 11111110 11111111111111111111111 = 1.11111111111111111111111 * two ^ 127
|-------------23--------------|

Even for floating points, when we add 1 to the data the number grows, until we reach our max, and when we subtract 1 our number shrinks, until we reach our min. So what happens when we **add 1 here?**

+ 1
= S 11111111 00000000000000000000000

We get **+∞**, which makes it a **seamless transition** in that sense because we should be expecting to get something bigger. We can't really add to infinity in any way that makes sense, so that's why if the fraction bits aren't zero here, we interpret it as NaN.

Too small:
Like before, let's try to find the smallest number that we can represent normally. We can't have the exponents all be zero, so we have the exponent bits be **00000001 = 1** which gives us our **smallest exponent of 1 - 127 = -126.**
Since we want to find the representation of the smallest, we set all the fraction bits to zero, which gives us our smallest number, that can be represented normally, to be:

S 00000001 00000000000000000000000 = 1.000... * two ^ -126

What if we wanted to get smaller? Unlike for 'too big,' we actually have a bit more wiggle room here. What if we didn't start with a leading 1? That's what the denormalized form is for. Let's subtract 1 from the floating point representation and see what happens.

- 1
= S 00000000 11111111111111111111111 = 0.11111111111111111111111 * two ^ -126

which is what we see in the table for how to interpret this case, for zeroes in the exponent bits and nonzero fraction bits. But now that begs the question: what is the smallest nonzero number we can represent?

smallest nonzero = S 00000000 00000000000000000000001 = 0.[22 zeroes]1 * two ^ -126
= 1.0 * two ^ (?)

Hint: we're working in binary so every time we move the decimal to the right, we subtract one from the exponent of two. This is also scientific notation, with the same rules for moving the decimal point, but in binary.

I hope you see here that it is also a **seamless** transition going from the smallest nonzero in normal form to even smaller.

Thank you to the staff and students for being awesome. Comments and/or critiques are encouraged, and I'd love to see you in my OHs!
Written by Anthony Winney.

datalab

~ An instructor (Shrey Shah) endorsed this note ~

Edit good note | 12 Updated 2 months ago by Anthony Winney

followup discussions *for lingering questions and comments*

Start a new followup discussion

Compose a new followup discussion