UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

UC Berkeley
Professor
Bora Nikolić

Floating Point

Berkeley
UNIVERSITY OF CALIFORNIA

cs61c.org

Basics &
Fixed Point

Quote of the day

"95% of the folks out there are completely clueless about floating-point."

– James Gosling, 1998-02-28

- Sun Fellow
- Java Inventor

Review of Numbers

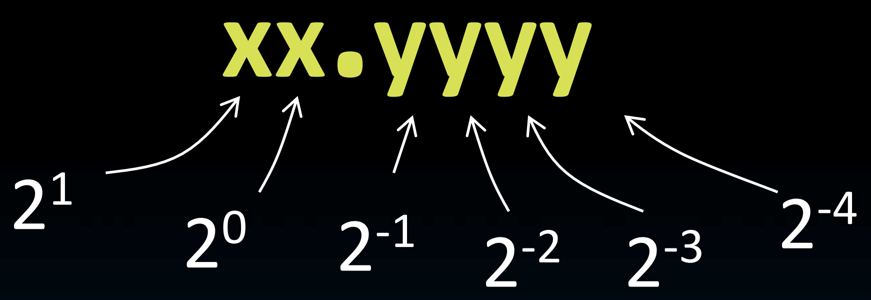
- Computers made to process numbers
- What can we represent in N bits?
  - 2N things, and no more! They could be…
  - Unsigned integers:
    - 0 to 2^N - 1
    - (for N=32, 2^N - 1 = 4,294,967,295)
  - Signed Integers (Two's Complement)
    - $-2^{(N-1)}$ to $2^{(N-1)}$ - 1
    - (for N=32, $2^{(N-1)}$ - 1 = 2,147,483,647)

What about other numbers?

- Very large numbers (sec/millennium)
  - 31,556,926,00010 (3.155692610 x 10^{10})
- Very small numbers? (Bohr radius)
  - 0.000000000052917710m (5.2917710 x 10^{-11})
- #s with both integer & fractional parts?
  - 1.5
- First consider #3.
  - …our solution will also help with 1 and 2.

Representation of Fractions

- **"Binary Point" like decimal point signifies boundary betw. integer and fractional parts:**
- **Example 6-bit representation**
- **$10.1010_2 = 1x2^1 + 1x2^{-1} + 1x2^{-3} = 2.625_{10}$**

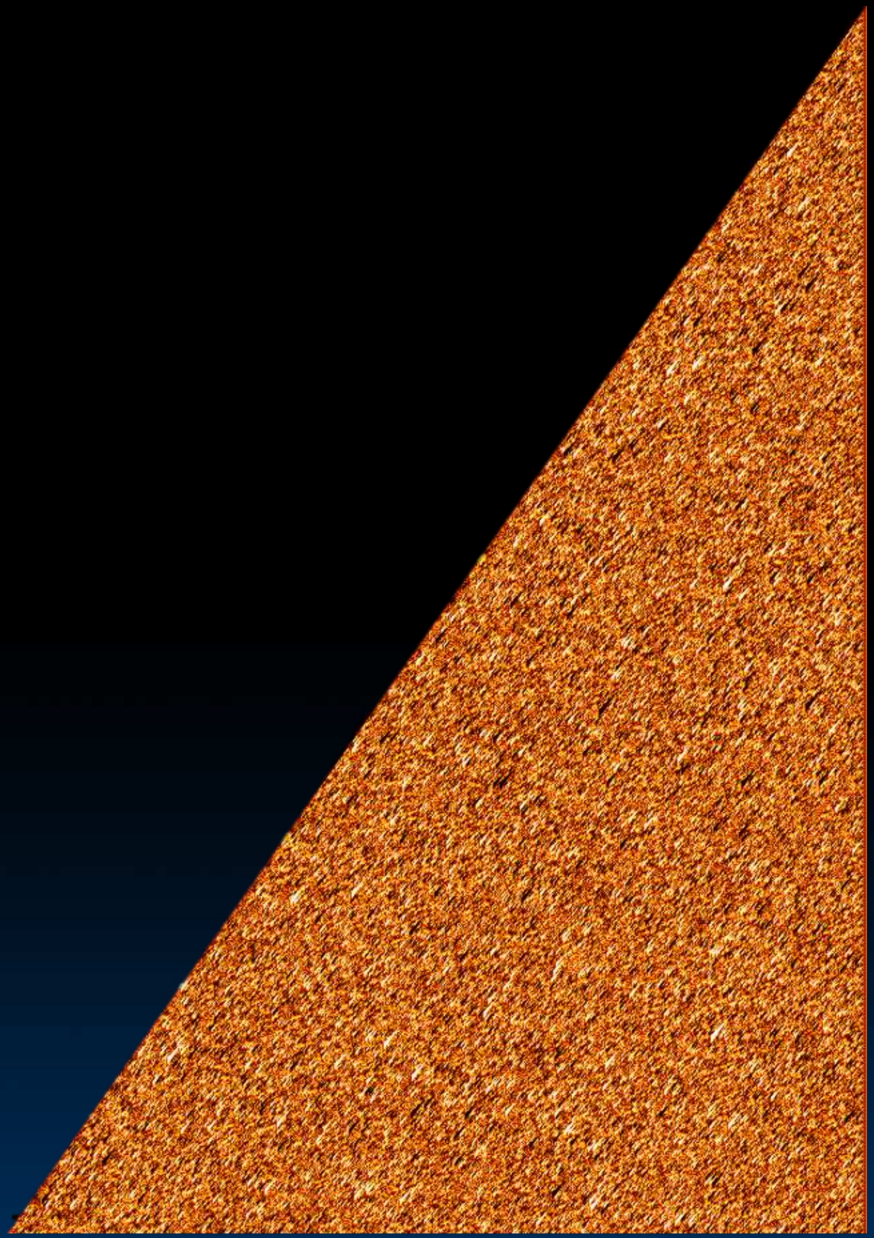


- If we assume "fixed binary point", range of 6-bit representations with this format:
  - 0 to 3.9375 (almost 4)

Garcia, Nikolić

Fractional Powers of 2

Mark Lu's "Binary Float Displayer"



i	2^{-i}	
0	1.0	1
1	0.5	1/2
2	0.25	1/4
3	0.125	1/8
4	0.0625	1/16
5	0.03125	1/32
6	0.015625	
7	0.0078125	
8	0.00390625	
9	0.001953125	
10	0.0009765625	
11	0.00048828125	
12	0.000244140625	
13	0.0001220703125	
14	0.00006103515625	
15	0.000030517578125	

Garcia, Nikolić

Representation of Fractions with Fixed Pt.

- What about addition and multiplication?
  - Addition is straightforward

```
  01.100   1.5₁₀
+ 00.100   0.5₁₀
  ------
  10.000   2.0₁₀
```

  - Multiplication a bit more complex:

```
       01.100   1.5₁₀
       00.100   0.5₁₀
       ------
       00 000
      000 00
     0110 0
    00000
   00000
   ----------
   0000110000
```

  - Where's the answer, 0.11?
    - Need to remember where point is…

Garcia, Nikolić

Floating Point

Representation of Fractions

- So far, in our examples we used a "fixed" binary point what we really want is to "float" the binary point. Why?
  - Floating binary point most effective use of our limited bits (and thus more accuracy in our number representation):
  - E.g., put 0.1640625 into binary. Represent as in 5-bits choosing where to put the binary point.
  - … 000000.001010100000…
  - Store these bits and keep track of the binary point 2 places to the left of the MSB.
  - Any other solution would lose accuracy!
- With floating point representation, each numeral carries an exponent field recording the whereabouts of its binary point.
- The binary point can be outside the stored bits, so very large and small numbers can be represented.

Garcia, Nikolić

Scientific Notation (in Decimal)

mantissa

exponent

$6.02_{ten} \times 10^{23}$

decimal point

radix (base)

- Normalized form: no leadings 0s (exactly one digit 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}

Garcia, Nikolić

Scientific Notation (in Binary)

mantissa

exponent

$$1.01_{two} \times 2^{-1}$$

"binary point"

radix (base)

- Computer arithmetic that supports it called floating point, because it represents numbers where the binary point is not fixed, as it is for integers
  - Declare such variable in C as `float`

Garcia, Nikolić

Floating Point Representation (1/2)

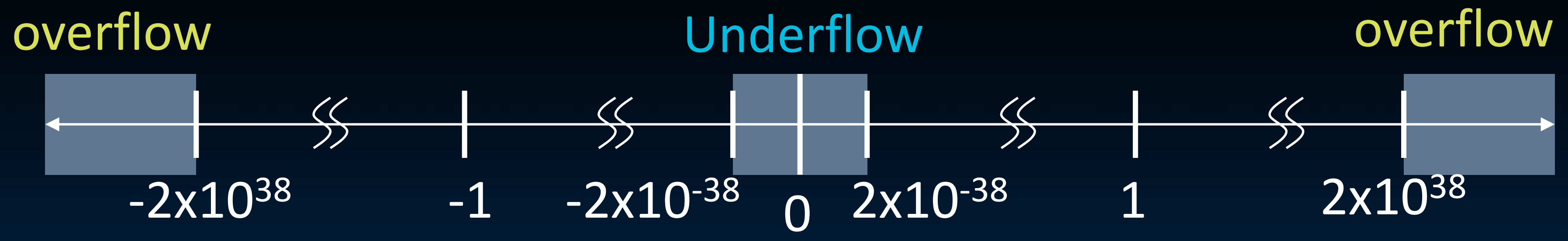
- Normal format: $+1.xxx...x_{two} * 2^{yyy...y}{}_{two}$
- Multiple of Word Size (32 bits)

- **S represents Sign**
- **Exponent represents y's**
- **Significand represents x's**
- **Represent numbers as small as 1.2×10^{-38} to as large as 3.4×10^{38}**

Berkeley
UNIVERSITY OF CALIFORNIA

Floating Point Representation (2/2)

- What if result too large?
  - (> 3.4×10^{38} , < -3.4×10^{38})
  - Overflow! → Exponent larger than represented in 8-bit Exponent field
- What if result too small?
  - (>0 and < 1.2×10^{-38} , <0 and > -1.2×10^{-38})
  - Underflow! → Negative exponent larger than represented in 8-bit Exponent field



- What would help reduce chances of overflow and/or underflow?

IEEE 754 Floating Point Standard (1/3)

- Single Precision (DP similar):

- Sign bit: 1 means negative, 0 means positive
- Significand:
  - To pack more bits, leading 1 implicit for normalized numbers
  - 1 + 23 bits single, 1 + 52 bits double
  - always true: 0 < Significand < 1 (for normalized numbers)
- Note: 0 has no leading 1, so reserve exponent value 0 just for number 0

IEEE 754 Floating Point Standard (2/3)

- IEEE 754 uses "biased exponent" representation.
  - Designers wanted FP numbers to be used even if no FP hardware; e.g., sort records with FP numbers using integer compares
  - Wanted bigger (integer) exponent field to represent bigger numbers.
  - 2's complement poses a problem (because negative numbers look bigger)
  - We're going to see that the numbers are ordered EXACTLY as in sign-magnitude
    - I.e., counting from binary odometer 00…00 up to 11…11 goes from 0 to +MAX to -0 to -MAX to 0

Garcia, Nikolić

IEEE 754 Floating Point Standard (3/3)

- Called Biased Notation, where bias is number subtracted to get real number
  - IEEE 754 uses bias of 127 for single prec.
  - Subtract 127 from Exponent field to get exponent value
- Summary (single precision, or fp32):

31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

- $(-1)^{S} \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
- Double precision identical, except exponent bias of 1023 (half, quad similar)…

Garcia, Nikolić

“Father” of the Floating point standard

- IEEE Standard 754 for Binary Floating-Point Arithmetic.

1989
ACM Turing Award Winner!

Prof. Kahan

www.cs.berkeley.edu/~wkahan/ieee754status/754story.html ☑

Special Numbers

Representation for ± ∞

- In FP, divide by 0 should produce ± ∞, not overflow.
- Why?
  - OK to do further computations with ∞
  - E.g., X/0 > Y may be a valid comparison
  - Ask math majors
- IEEE 754 represents ± ∞
  - Most positive exponent reserved for ∞
  - Significands all zeroes

Garcia, Nikolić

Representation for 0

- Represent 0?
  - exponent all zeroes
  - significand all zeroes
  - What about sign? Both cases valid.

```
+0: 0 00000000 00000000000000000000000
-0: 1 00000000 00000000000000000000000
```

Berkeley
UNIVERSITY OF CALIFORNIA

Garcia, Nikolić

Special Numbers

- What have we defined so far? (Single Precision)

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	???

- Professor Kahan had clever ideas; "Waste not, want not"
  - Wanted to use Exp=0,255 & Sig!=0

Representation for Not a Number

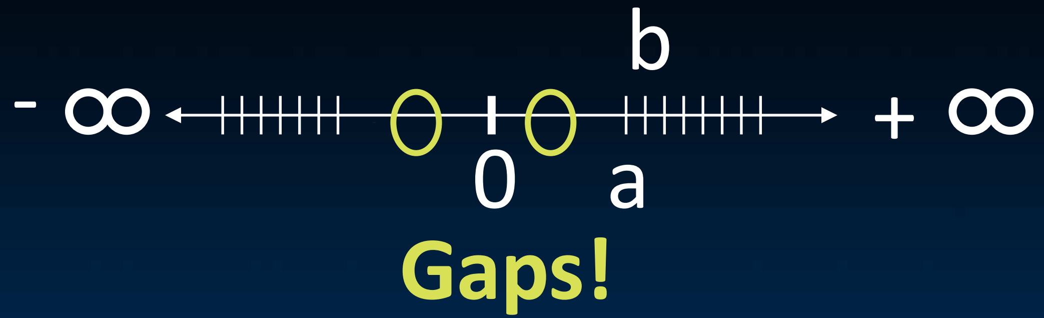
- What do I get if I calculate sqrt(-4.0) or 0/0?
  - If ∞ not an error, these shouldn't be either
  - Called Not a Number (NaN)
  - Exponent = 255, Significand nonzero
- Why is this useful?
  - Hope NaNs help with debugging?
  - They contaminate: op(NaN, X) = NaN
  - Can use the significand to identify which!

Garcia, Nikolić

Representation for Denorms (1/2)

- Problem: There's a gap among representable FP numbers around 0
  - Smallest representable pos num:
    - $a = 1.0\ldots_2 * 2^{-126} = 2^{-126}$
  - Second smallest representable pos num:
    - $b = 1.000\ldots\ldots1_2 * 2^{-126}$
      $= (1 + 0.00\ldots1_2) * 2^{-126}$
      $= (1 + 2^{-23}) * 2^{-126}$
      $= 2^{-126} + 2^{-149}$
  - $a - 0 = 2^{-126}$
  - $b - a = 2^{-149}$

Normalization and implicit 1 is to blame!



Representation for Denorms (2/2)

- Solution:
  - We still haven't used Exponent = 0, Significand nonzero
  - DEnormalized number: no (implied) leading 1, implicit exponent = -126.
  - Smallest representable pos num:
    - $a = 2^{-149}$
  - Second smallest representable pos num:
    - $b = 2^{-148}$

Garcia, Nikolić

Special Numbers Summary

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	nonzero	Denorm
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	NaN

Examples, Discussion

Example

- What is the decimal equivalent of:

1	1000 0001	111 0000 0000 0000 0000 0000
S	Exponent	Significand

$$(-1)^{S} \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$$

$$(-1)^{1} \times (1 + .111)_2 \times 2^{(129-127)}$$

$$-1 \times (1.111)_2 \times 2^{(2)}$$

$$-111.1_2$$

$$-7.5_{10}$$

Garcia, Nikolić

Example: Representing 1/3

- 1/3
  - $= 0.33333\ldots_{10}$
  - $= 0.25 + 0.0625 + 0.015625 + 0.00390625 + \ldots$
  - $= 1/4 + 1/16 + 1/64 + 1/256 + \ldots$
  - $= 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + \ldots$
  - $= 0.0101010101\ldots_2 * 2^0$
  - $= 1.0101010101\ldots_2 * 2^{-2}$
  - **Sign**: 0
  - **Exponent** = -2 + 127 = 125 = 01111101
  - **Significand** = 0101010101…

0	0111 110	1 0101 0101 0101 0101 0101 010

Garcia, Nikolić

Understanding the Significand (1/2)

- Method 1 (Fractions):
  - In decimal: 0.340_{10} $\Rightarrow 340_{10}/1000_{10}$
    $\Rightarrow 34_{10}/100_{10}$
  - In binary: 0.110_2 $\Rightarrow 110_2/1000_2 = 6_{10}/8_{10}$
    $\Rightarrow 11_2/100_2 = 3_{10}/4_{10}$
  - Advantage: less purely numerical, more thought oriented; this method usually helps people understand the meaning of the significand better

Garcia, Nikolić

Understanding the Significand (2/2)

- Method 2 (Place Values):
  - Convert from scientific notation
  - In decimal:

$$1.6732 = (1x10^{0}) + (6x10^{-1}) + (7x10^{-2}) + (3x10^{-3}) + (2x10^{-4})$$

  - In binary:

$$1.1001 = (1x2^{0}) + (1x2^{-1}) + (0x2^{-2}) + (0x2^{-3}) + (1x2^{-4})$$

  - Interpretation of value in each position extends beyond the decimal/binary point
  - Advantage: good for quickly calculating significand value; use this method for translating FP numbers

☑

Floating Point Discussion

Floating Point Fallacy

- FP add associative?
  - $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, and $z = 1.0$
  - $x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0)$
    $= -1.5 \times 10^{38} + (1.5 \times 10^{38}) = \underline{0.0}$
  - $(x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0$
    $= (0.0) + 1.0 = \underline{1.0}$
- Therefore, Floating Point add is not associative!
  - Why? FP result approximates real result!
  - This example: 1.5×10^{38} is so much larger than 1.0 that $1.5 \times 10^{38} + 1.0$ in floating point representation is still 1.5×10^{38}

Garcia, Nikolić

Precision and Accuracy

Don't confuse these two terms!

- Precision is a count of the number bits in used to represent a value.
- Accuracy is the difference between the actual value of a # and its computer representation.
- High precision permits high accuracy but doesn't guarantee it.
  - It is possible to have high precision but low accuracy.
- Example: `float pi = 3.14;`
  - **`pi`** will be represented using all 24 bits of the significant (highly precise), but is only an approximation (not accurate).

Rounding

- When we perform math on real numbers, we have to worry about rounding to fit the result in the significant field.
- The FP hardware carries two extra bits of precision, and then round to get the proper value
- Rounding also occurs when converting:

  double to a single precision value, or

  floating point number to an integer

Garcia, Nikolić

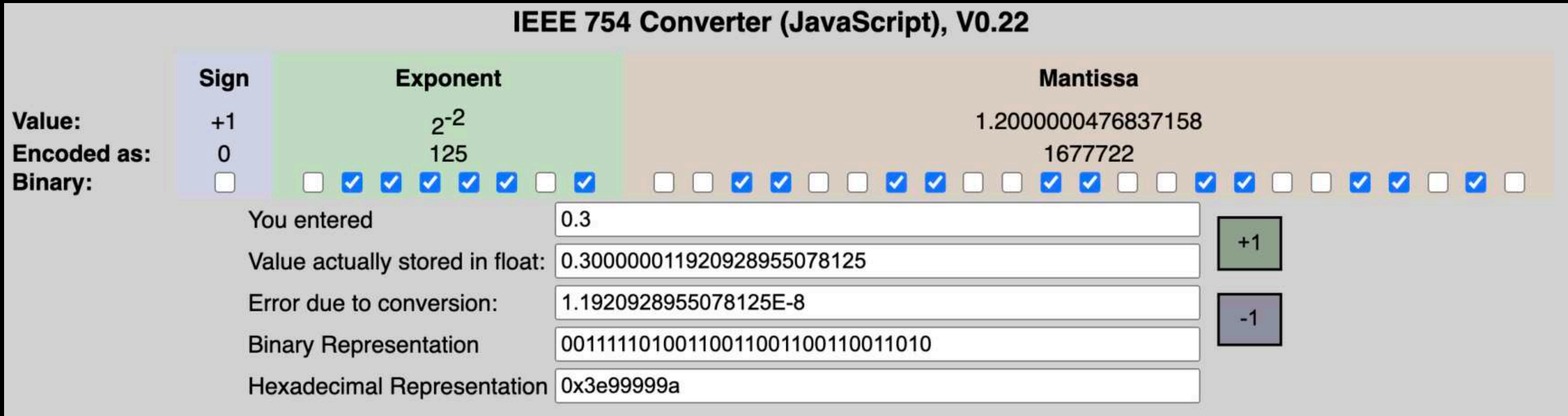
IEEE FP Rounding Modes

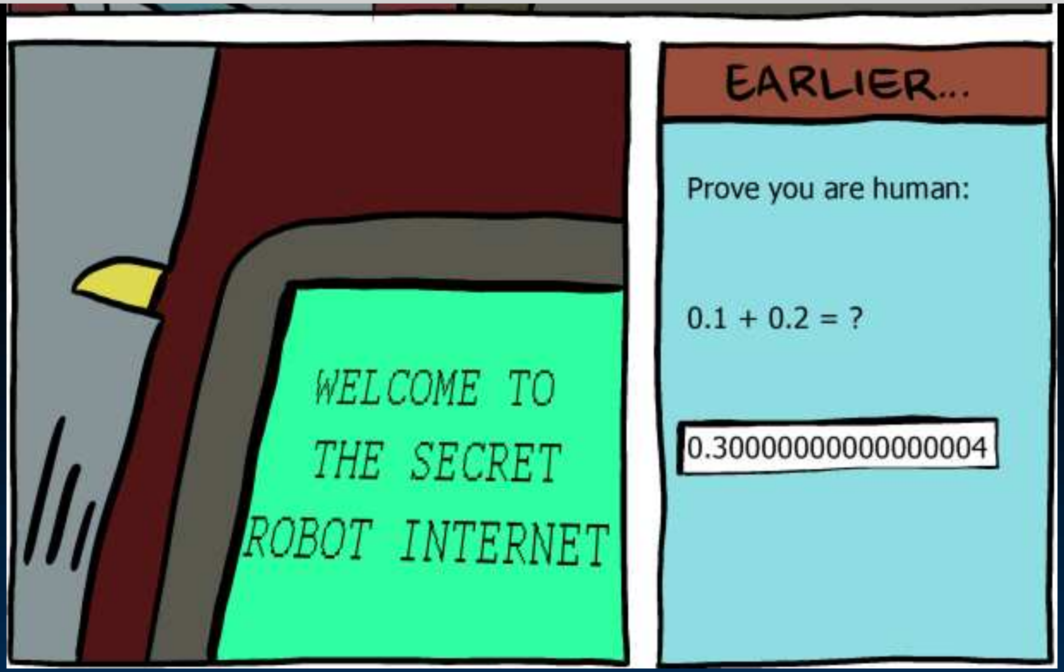
- Round towards + ∞
  - ALWAYS round "up": 2.001 → 3, -2.001 → -2
- Round towards - ∞
  - ALWAYS round "down": 1.999 → 1, -1.999 → -2
- Truncate
  - Just drop the last bits (round towards 0)
- Unbiased (default mode). Midway? Round to even
  - Normal rounding, almost: 2.4 → 2, 2.6 → 3, 2.5 → 2, 3.5 → 4
  - Round like you learned in grade school (nearest int)
  - Except if the value is right on the borderline, in which case we round to the nearest EVEN number
  - Ensures fairness on calculation
  - This way, half the time we round up on tie, the other half time we round down. Tends to balance out inaccuracies

Examples in decimal (but, of course, IEEE754 in binary)

Garcia, Nikolić

Now you know why you see these errors…





Saturday Morning Breakfast Comics

www.smbc-comics.com/comic/2013-06-05

FP Addition

- More difficult than with integers
- Can't just add significands
- How do we do it?
  - De-normalize to match exponents
  - Add significands to get resulting one
  - Keep the same exponent
  - Normalize (possibly changing exponent)
- Note: If signs differ, just perform a subtract instead.

Garcia, Nikolić

Casting floats to ints and vice versa

```
(int) floating_point_expression
```

Coerces and converts it to the nearest integer (C uses truncation)

```
i = (int) (3.14159 * f);
```

```
(float) integer_expression
```

converts integer to nearest floating point

```
f = f + (float) i;
```


Garcia, Nikolić

int → float → int

```
if (i == (int)((float) i)) {
  printf("true");
}
```

- Will not always print "**true**"
- Most large values of integers don't have exact floating point representations!
- What about **double**?

Garcia, Nikolić

float → int → float

```
if (f == (float)((int) f)) {
 printf("true");
}
```

- Will not always print "**true**"
- Small floating point numbers (<1) don't have integer representations
- For other numbers, rounding errors

☑

Other Floating Point Representations

Double Precision Fl. Pt. Representation

- **binary64**: Next Multiple of Word Size (64 bits)

31 30 20 19 0

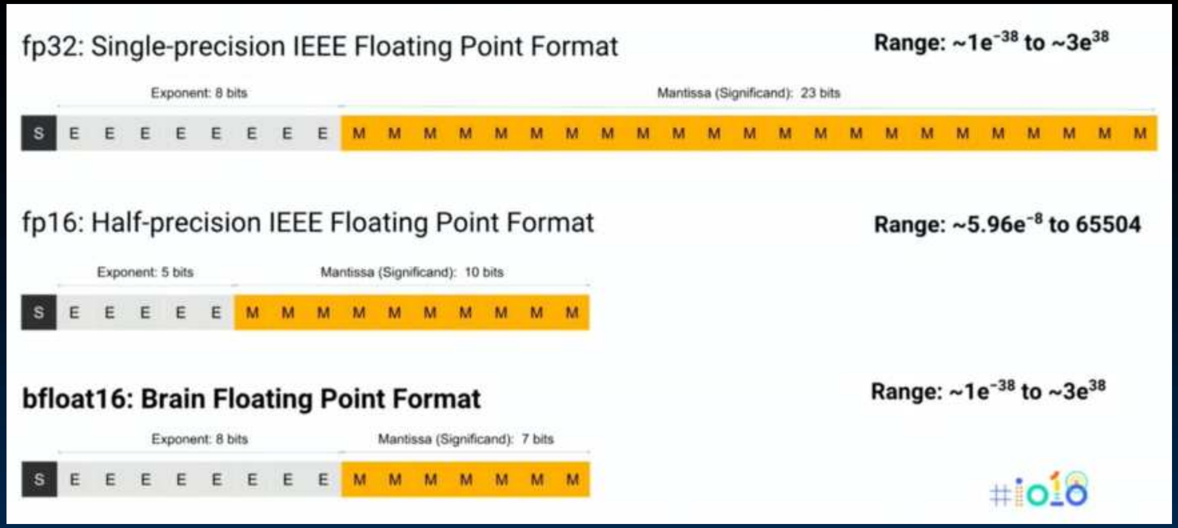
S	Exponent	Significand
1 bit	11 bits	20 bits

Significand (cont'd)
32 bits

- Double Precision (vs. Single Precision)
  - C variable declared as `double`
  - Represent numbers almost as small as 2.0×10^{-308} to almost as large as 2.0×10^{308}
  - But primary advantage is greater accuracy due to larger significand

Other Floating Point Representations

- Quad-Precision? Yep! (128 bits) “binary128”
  - Unbelievable range, precision (accuracy)
  - 15 exponent bits, 112 significand bits
- Oct-Precision? Yep! “binary256”
  - 19 exponent bits, 236 significant bits
- Half-Precision? Yep! “binary16” or “fp16”
  - 1/5/10 bits
- Half-Precision?
  Yep! “bfloat16”
  - Competing with fp16
  - Same range as fp32!
  - Used for faster ML



en.wikipedia.org/wiki/Floating_point

Garcia, Nikolić

Floating Point Soup

FP32

31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

FP16

15	14 … 10	9 … 0
S	Exponent	Significand
1 bit	5 bits	10 bits

BFLOAT16

15	14 … 7	6 … 0
S	Exponent	Significand
1 bit	8 bits	7 bits

TF32

18	17 … 10	9 … 0
S	Exponent	Significand
1 bit	8 bits	10 bits

Garcia, Nikolić

Who Uses What in Domain Accelerators?

Accelerator	int4	int8	int16	fp16	bf16	fp32	tf32
Google TPU v1		x					
Google TPU v2					x		
Google TPU v3					x		
Nvidia Volta TensorCore	x	x		x			
Nvidia Ampere TensorCore	x	x	x	x	x	x	x
Nvidia DLA		x	x	x			
Intel AMX		x			x		
Amazon AWS Inferentia		x		x	x		
Qualcomm Hexagon		x					
Huawei Da Vinci		x		x			
MediaTek APU 3.0		x	x	x			
Samsung NPU		x					
Tesla NPU		x					

Garcia, Nikolić

Unum

- Everything so far has had a fixed set of bits for Exponent and Significant
  - What if they were variable?
  - Add a "u-bit" to tell whether number is exact or range
  - "Promises to be to floating point what floating point is to fixed point"
- Claims to save power!

Dr. John Gustafson

Conclusion

- Floating Point lets us:
  - Represent numbers containing both integer and fractional parts; makes efficient use of available bits.
  - Store approximate values for very large and very small #s.
- IEEE 754 Floating Point Standard is most widely accepted attempt to standardize interpretation of such numbers
  - Every computer since ~1997 follows these conventions)
- Summary (single precision, or fp32):

31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

$(-1)^{S} \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

Can store NaN, ± ∞ ☑

Exponent tells Significand how much (2^i) to count by (…, 1/4, 1/2, 1, 2, …)

Garcia, Nikolić