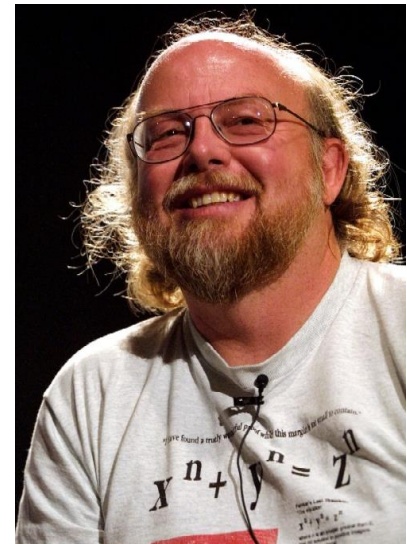


Quote of the day

**“95% of the
folks out there are
completely clueless
about floating-point.”**

**James Gosling
Sun Fellow
Java Inventor
1998-02-28**



Review of Numbers

- Computers are made to deal with numbers
- What can we represent in N bits?
 - 2^N 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 $N=32$, $2^{(N-1)} = 2,147,483,648$)

What about other numbers?

1. Very large numbers? (seconds/millennium)
 $\Rightarrow 31,556,926,000_{10} (3.1556926_{10} \times 10^{10})$
2. Very small numbers? (Bohr radius)
 $\Rightarrow 0.0000000000529177_{10}\text{m} (5.29177_{10} \times 10^{-11})$
3. Numbers with both integer & fractional parts?
 $\Rightarrow 1.5$

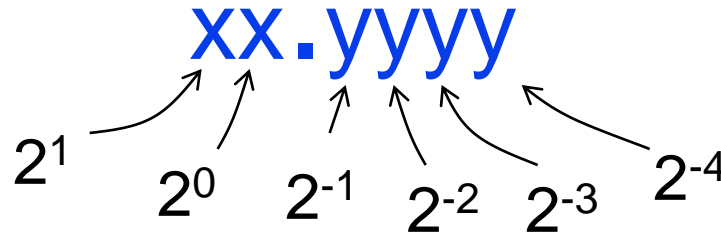
First consider #3.

...our solution will also help with 1 and 2.

Representation of Fractions

“Binary Point” like decimal point signifies boundary between integer and fractional parts:

Example 6-bit representation:



$$10.1010_2 = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{10}$$

If we assume “fixed binary point”, range of 6-bit representations with this format:

0 to 3.9375 (almost 4)

Fractional Powers of 2

Mark Lu's "Binary Float Displayer"

<http://inst.eecs.berkeley.edu/~marklu/bfd/?n=1000>



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	

Representation of Fractions with Fixed Pt.

What about addition and multiplication?

Addition is straightforward:

01.100	1.5_{10}
+ 00.100	0.5_{10}
<hr/>	
10.000	2.0_{10}

01.100	1.5_{10}
00.100	0.5_{10}
<hr/>	
00 000	

Multiplication a bit more complex:

000 00
0110 0
00000
00000
<hr/>
0000110000

Where's the answer, 0.11? (need to remember where point is)


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):

example: 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 rep., each numeral carries a 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.

Scientific Notation (in Decimal)

The diagram shows the expression $6.02_{10} \times 10^{23}$ with four labels and arrows pointing to its parts: 'mantissa' points to '6.02', 'decimal point' points to the dot in '6.02', 'radix (base)' points to the subscript '10' in '10', and 'exponent' points to the superscript '23' in '10²³'.

- **Normalized form: no leading 0s**
(exactly one digit to left of decimal point)
- **Alternatives to representing 1/1,000,000,000**
 - **Normalized:** 1.0×10^{-9}
 - **Not normalized:** $0.1 \times 10^{-8}, 10.0 \times 10^{-10}$

Scientific Notation (in Binary)

mantissa exponent

 1.01_{two} x 2⁻¹

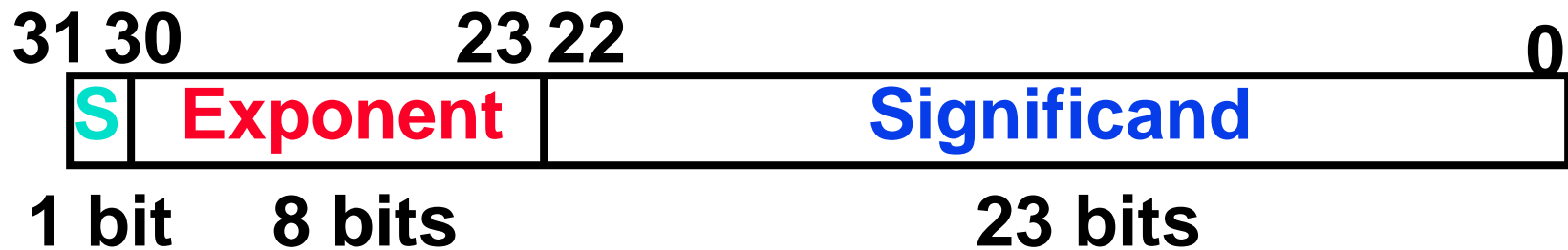
 ↑ ↙

 “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`

Floating Point Representation (1/2)

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



- S represents Sign
Exponent represents y's
Significand represents x's
- Represent numbers as small as 2.0×10^{-38} to as large as 2.0×10^{38}

Floating Point Representation (2/2)

- What if result too large?

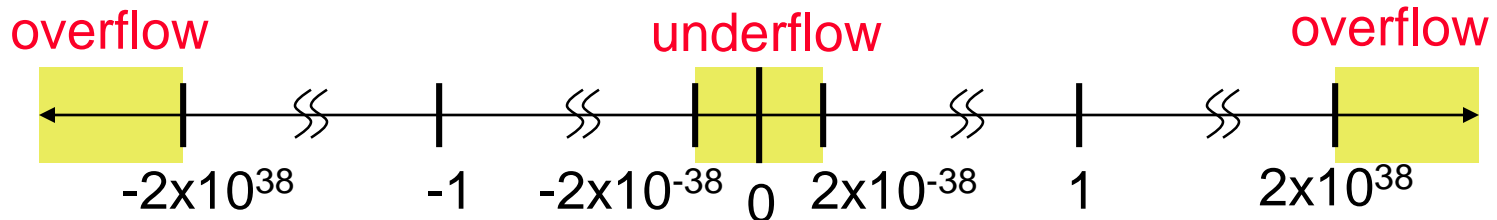
($> 2.0 \times 10^{38}$, $< -2.0 \times 10^{38}$)

- **Overflow!** \Rightarrow Exponent larger than represented in 8-bit Exponent field

- What if result too small?

(>0 & $< 2.0 \times 10^{-38}$, <0 & $> -2.0 \times 10^{-38}$)

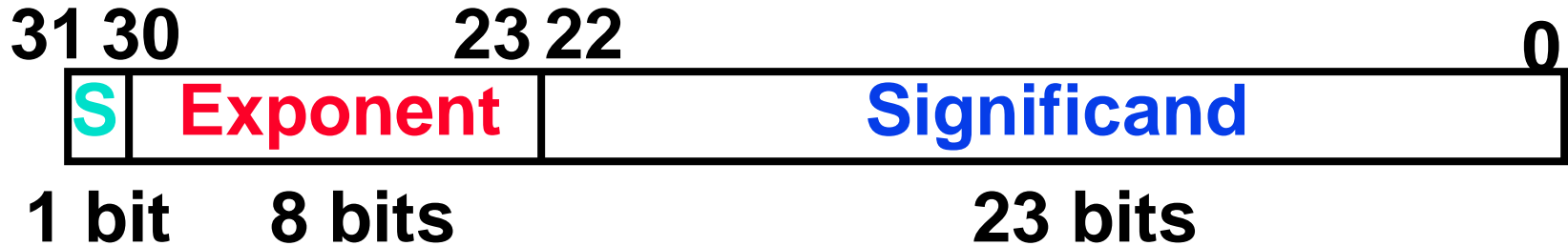
- **Underflow!** \Rightarrow Negative exponent larger than represented in 8-bit Exponent field



- 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: $0 < \text{Significand} < 1$
(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

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 actual value for exponent
 - 1023 is bias for double precision

• Summary (single precision):



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

- Double precision identical, except with exponent bias of 1023 (half, quad similar)

“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

Representation for $\pm \infty$

- In FP, divide by 0 should produce $\pm \infty$, 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 $\pm \infty$
 - Most positive exponent reserved for ∞
 - Significands all zeroes

Representation for 0

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

+0: 0 00000000 00000000000000000000000000000000

-0: 1 00000000 00000000000000000000000000000000

Special Numbers

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

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

- Professor Kahan had clever ideas;
“Waste not, want not”
 - Wanted to use $\text{Exp}=0,255$ & $\text{Sig}\neq 0$

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!

Representation for Denorms (1/2)

- **Problem: There's a gap among representable FP numbers around 0**

- **Smallest representable pos num:**

$$a = 1.0 \dots_2 * 2^{-126} = 2^{-126}$$

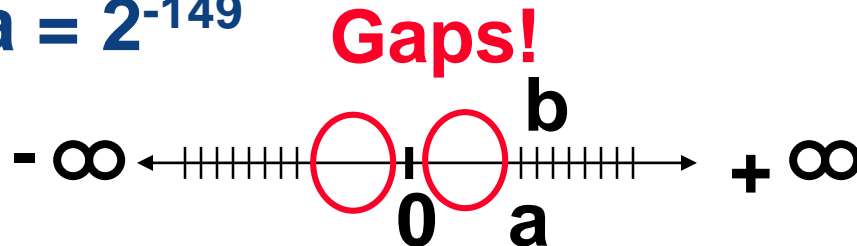
- **Second smallest representable pos num:**

$$\begin{aligned} b &= 1.000 \dots 1_2 * 2^{-126} \\ &= (1 + 0.00 \dots 1_2) * 2^{-126} \\ &= (1 + 2^{-23}) * 2^{-126} \\ &= 2^{-126} + 2^{-149} \end{aligned}$$

$$a - 0 = 2^{-126}$$

$$b - a = 2^{-149}$$

**Normalization
and implicit 1
is to blame!**



Representation for Denorms (2/2)

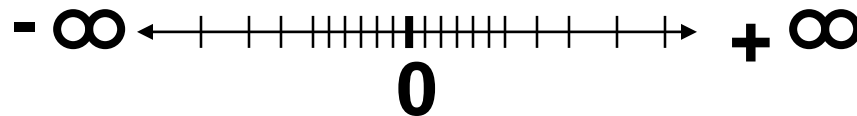
- **Solution:**

- 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:

$$b = 2^{-148}$$



Special Numbers Summary

- Reserve exponents, significands:

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

Conclusion

• Floating Point lets us:

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

- Represent numbers containing both integer and fractional parts; makes efficient use of available bits.
- Store **approximate** values for very large and very small #s.

Can
store
NaN,
 $\pm \infty$

- **IEEE 754 Floating Point Standard** is most widely accepted attempt to standardize interpretation of such numbers (Every desktop or server computer sold since ~1997 follows these conventions)

• Summary (single precision):



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

- **Double precision identical, except with exponent bias of 1023 (half, quad similar)**

Bonus slides

- These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

Bonus

Example: Converting Binary FP to Decimal

0	0110 1000	101 0101 0100 0011 0100 0010
---	-----------	------------------------------

- **Sign:** 0 → positive
- **Exponent:**
 - 0110 1000_{two} = 104_{ten}
 - Bias adjustment: 104 - 127 = -23
- **Significand:**
$$1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + \dots$$
$$= 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22}$$
$$= 1.0 + 0.666115$$
- **Represents:** 1.666115_{ten} * 2⁻²³ ~ 1.986 * 10⁻⁷
(about 2/10,000,000)

Example: Converting Decimal to FP

-2.340625 x 10¹

1. Denormalize: -23.40625

2. Convert integer part:

$$23 = 16 + (7 = 4 + (3 = 2 + (1))) = 10111_2$$

3. Convert fractional part:

$$.40625 = .25 + (.15625 = .125 + (.03125)) = .01101_2$$

4. Put parts together and normalize:

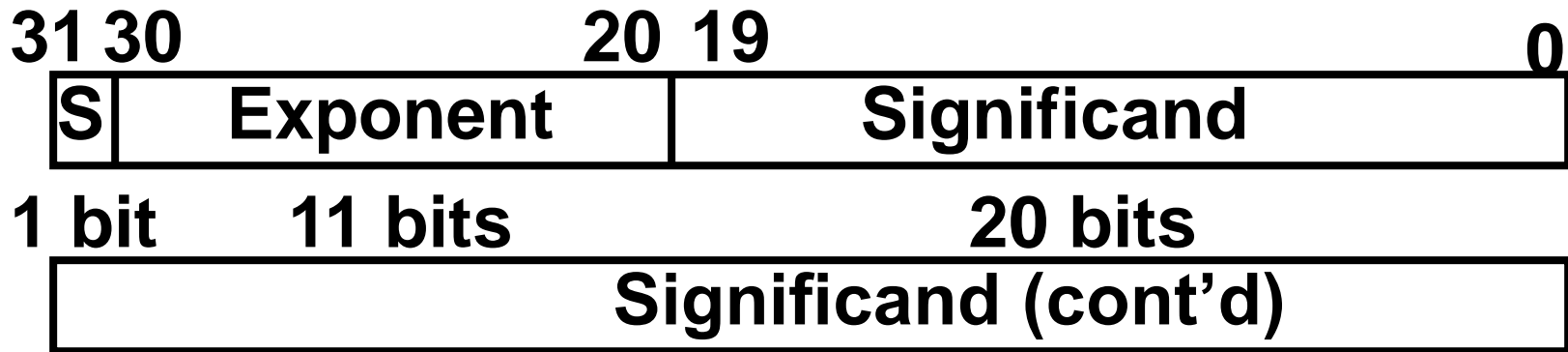
$$10111.01101 = 1.011101101 \times 2^4$$

5. Convert exponent: $127 + 4 = 10000011_2$

1	1000 0011	011 1011 0100 0000 0000 0000
---	-----------	------------------------------

Double Precision Fl. Pt. Representation

- Next Multiple of Word Size (64 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

QUAD Precision Fl. Pt. Representation

- **Next Multiple of Word Size (128 bits)**
 - Unbelievable **range** of numbers
 - Unbelievable **precision** (accuracy)
- **IEEE 754-2008 “binary128” standard**
 - Has 15 exponent bits and 112 significand bits (113 precision bits)
- **Oct-Precision?**
 - Some have tried, no real traction so far
- **Half-Precision?**
 - Yep, “binary16”: 1/5/10

en.wikipedia.org/wiki/Floating_point

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

Understanding the Significand (2/2)

- **Method 2 (Place Values):**
 - Convert from scientific notation
 - In decimal: $1.6732 = (1 \times 10^0) + (6 \times 10^{-1}) + (7 \times 10^{-2}) + (3 \times 10^{-3}) + (2 \times 10^{-4})$
 - In binary: $1.1001 = (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$
 - 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

Precision and Accuracy

Don't confuse these two terms!

Precision is a count of the number bits in a computer word used to represent a value.

Accuracy is a measure of the difference between the actual value of a number 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

IEEE FP Rounding Modes

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

- **Round towards $+\infty$**
 - ALWAYS round “up”: $2.001 \rightarrow 3$, $-2.001 \rightarrow -2$
- **Round towards $-\infty$**
 - ALWAYS round “down”: $1.999 \rightarrow 1$, $-1.999 \rightarrow -2$
- **Truncate**
 - Just drop the last bits (round towards 0)
- **Unbiased (default mode). Midway? Round to even**
 - Normal rounding, almost: $2.4 \rightarrow 2$, $2.6 \rightarrow 3$, $2.5 \rightarrow 2$, $3.5 \rightarrow 4$
 - 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

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.**

MIPS Floating Point Architecture (1/4)

- **MIPS has special instructions for floating point operations:**
 - **Single Precision:**
`add.s, sub.s, mul.s, div.s`
 - **Double Precision:**
`add.d, sub.d, mul.d, div.d`
- **These instructions are far more complicated than their integer counterparts. They require special hardware and usually they can take much longer to compute.**

MIPS Floating Point Architecture (2/4)

- **Problems:**

- **It's inefficient to have different instructions take vastly differing amounts of time.**
- **Generally, a particular piece of data will not change from FP to int, or vice versa, within a program. So only one type of instruction will be used on it.**
- **Some programs do no floating point calculations**
- **It takes lots of hardware relative to integers to do Floating Point fast**

MIPS Floating Point Architecture (3/4)

- **1990 Solution: Make a completely separate chip that handles only FP.**
- **Coprocessor 1: FP chip**
 - contains 32 32-bit registers: \$f0, \$f1, ...
 - most registers specified in .s and .d instruction refer to this set
 - separate load and store: lwc1 and swc1 (“load word coprocessor 1”, “store ...”)
 - Double Precision: by convention, even/odd pair contain one DP FP number: \$f0/\$f1, \$f2/\$f3, ... , \$f30/\$f31

MIPS Floating Point Architecture (4/4)

- **1990 Computer actually contains multiple separate chips:**
 - **Processor:** handles all the normal stuff
 - **Coprocessor 1:** handles FP and only FP;
 - **more coprocessors?... Yes, later**
 - **Today, cheap chips may leave out FP HW**
- **Instructions to move data between main processor and coprocessors:**
 - **`mfc0, mtc0, mfc1, mtc1, etc.`**
- **Appendix pages A-70 to A-74 contain many, many more FP operations.**

Example: Representing 1/3 in MIPS

- **1/3**

$$= 0.33333..._{10}$$

$$= 0.25 + 0.0625 + 0.015625 + 0.00390625 + ...$$

$$= 1/4 + 1/16 + 1/64 + 1/256 + ...$$

$$= 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + ...$$

$$= 0.0101010101..._2 * 2^0$$

$$= 1.0101010101..._2 * 2^{-2}$$

- **Sign: 0**

- **Exponent = $-2 + 127 = 125 = 01111101$**

- **Significand = 0101010101...**

0	0111 1101	0101 0101 0101 0101 0101 010
---	-----------	------------------------------

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;
```


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?

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

Floating Point Fallacy

- FP add associative: FALSE!

- $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, 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}

Peer Instruction

1	1000 0001	111 0000 0000 0000 0000 0000
---	-----------	------------------------------

What is the decimal equivalent of the floating pt # above?

- a) $-7 * 2^{129}$
- b) -3.5
- c) -3.75
- d) -7
- e) -7.5

Peer Instruction

1. Converting float -> int -> float produces same float number
2. Converting int -> float -> int produces same int number
3. FP add is associative:
 $(x+y)+z = x+(y+z)$

	ABC
1:	FFF
2:	FFT
3:	FTF
4:	FTT
5:	TFF
6:	TFT
7:	TTF
8:	TTT

Peer Instruction

- Let $f(1, 2)$ = # of floats between 1 and 2
- Let $f(2, 3)$ = # of floats between 2 and 3

1:	$f(1, 2)$	$<$	$f(2, 3)$
2:	$f(1, 2)$	$=$	$f(2, 3)$
3:	$f(1, 2)$	$>$	$f(2, 3)$