CSE 31
Computer Organization

Lecture 18 – Floating Point Numbers (2)

Announcement

- Lab #8 this week
  - Due in one week
- HW #5 (from zyBooks)
  - Due Monday (11/5)
- Project #2 out this Friday
  - Due Monday (12/3)
    - Don’t start late, you won’t have time!
- Reading assignment
  - Chapter 1.6, 6.1 – 6.7 of zyBooks (Reading Assignment #5)
    - Make sure to do the Participation Activities
    - Due Monday (11/5, extended)

Announcement

- Midterm Exam 2
  - 11/19 (Monday, in lecture) Not 11/7 as scheduled
  - Lectures #8 - #18
  - HW #3 - #5
  - Practice exam in CatCourses
  - Closed book
  - 1 sheet of note (8.5” x 11”)
  - MIPS reference sheet will be provided

Review

- 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 desktop or server computer sold since ~1997 follows these conventions)

single precision:

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 with exponent bias of 1023 (half, quad similar)

Example: Representing 1/3 in MIPS

- 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 1101	0101 0101 0101 0101 0101 010

Floating Point Fallacy

- FP add associative?
  - $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, and $z = 1.0$
  - $x + (y + z) \quad = -1.5\text{x}10^{38} + (1.5\text{x}10^{38} + 1.0)$
    $= -1.5\text{x}10^{38} + (1.5\text{x}10^{38}) = \underline{\mathbf{0.0}}$
  - $(x + y) + z \quad = (-1.5\text{x}10^{38} + 1.5\text{x}10^{38}) + 1.0$
    $= (0.0) + 1.0 = \underline{\mathbf{1.0}}$
- <u>Therefore, Floating Point add is NOT associative!</u>
  - Why?
    - FP result <u>approximates</u> 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}

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).

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

0	1111 1111	0000 0000 0000 0000 0000 000

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
```

Special Numbers

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

Exponent	Significand	Object
0	0	0
0	Nonzero	???
1-254	Anything	+/- FP #
255	0	+/- ∞
255	Nonzero	???

- "Waste not, want not"
  - We'll talk about Exp=0,255 & Sig!=0 next

Representation for "Not a Number"

- What do you get if you calculate `sqrt(-4.0)` or `0/0`?
  - If ∞ is 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

Representation for Denorms (1/2)

- Problem: There's a gap among representable FP numbers around 0
  - Smallest representable positive number:

    a = $1.0\ldots_2 * 2^{-126} = 2^{-126}$ (exp = 1, sig = 0)
  - Second smallest representable pos num:

    b = $1.000\ldots\ldots1_2 * 2^{-126}$ (exp = 1, sig = 1)

    $= (1 + 0.00\ldots1_2) * 2^{-126}$

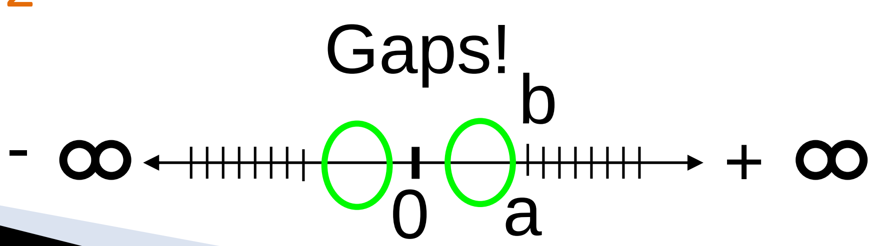
    $= (1 + 2^{-23}) * 2^{-126}$

    $= 2^{-126} + 2^{-149}$

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

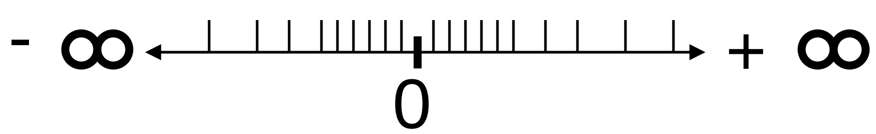
    $b - a = 2^{-149}$

Normalization and implicit 1 is to blame!



Representation for Denorm (2/2)

- Solution:
  - We still haven't used Exponent=0, Significand nonzero
  - Denormalized number: no (implied) leading 1, implicit exponent = -126 (0 – 127 + 1)
    - $(-1)^S \times (\text{Significand}) \times 2^{(-126)}$
  - Smallest representable pos num:
    - $a = 2^{-149}$ (sig = 1)
  - Second smallest representable pos num:
    - $b = 2^{-148}$ (sig = 2)



Special Numbers Summary

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	Nonzero	Denorm
1-254	Anything	+/- FP #
255	0	+/- ∞
255	Nonzero	NaN

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

- Halfway between two floating point values (rounding bits read <u>10)</u>? Choose from the following:
  - Round towards + ∞
    - Round "up": 1.01 <u>10</u> → 1.10 , -1.01 <u>10</u> → -1.01
  - Round towards - ∞
    - Round "down": 1.01 <u>10</u> → 1.01, -1.01 <u>10</u> → -1.10
- Truncate
  - Just drop the extra bits (round towards 0)
- Unbiased (default mode). Round to nearest EVEN number
  - Half the time we round up on tie, the other half time we round down. Tends to balance out inaccuracies.
  - In binary, even means least significant bit is 0.
- Otherwise, not halfway (<u>00</u>, <u>01</u>, <u>11</u>)! Just round to the nearest float.

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

Quiz 1:

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

Quiz 1:

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)`

1. 3.14 -> 3 -> 3

2. 32 bits for signed int,
   but 24 for FP mantissa?

3. x = biggest pos #,
   y = -x, z = 1 (x != inf)

ABC
1: FFF
2: FFT
3: FTF
4: FTT
5: TFF

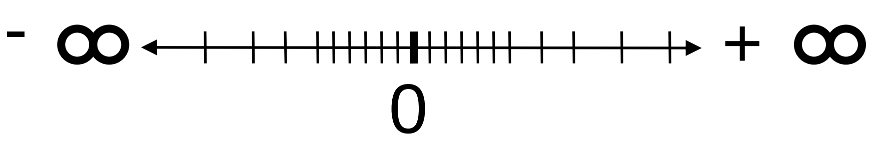
Quiz 2:

- 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: $f(1,2) > f(2,3)$

Quiz 2:

- 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: $f(1,2) > f(2,3)$

Summary

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	Nonzero	Denorm
1-254	Anything	+/- FP #
255	0	+/- ∞
255	Nonzero	NaN

- 4 Rounding modes (default: unbiased)
- MIPS Fl ops complicated, expensive