Floating Point I

ACKNOWLEDGEMENT: These slides have been modified by your your CMPT 295 instructor and CS:APP Textbook authors. However, please report all mistakes to your instructor.

Number Representation Revisited

- What can we represent in one word?
  - Signed and Unsigned Integers
  - Characters (ASCII)
  - Addresses
- How do we encode the following:
  - Real numbers (*e.g.* 3.14159)
  - Very large numbers (*e.g.* 6.02×10^{23})
  - Very small numbers (*e.g.* 6.626×10^{-34})
  - Special numbers (*e.g.* ∞, NaN)

Floating Point

Number Representation Really Matters

- **1991:** Patriot missile targeting error
  - clock skew due to conversion from integer to floating point
- **1996:** Ariane 5 rocket exploded ($1 billion)
  - overflow converting 64-bit floating point to 16-bit integer
- **2000:** Y2K problem
  - limited (decimal) representation: overflow, wrap-around
- **2038:** Unix epoch rollover
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to TMin in 2038
- **Other related bugs:**
  - 1982: Vancouver Stock Exchange 10% error in less than 2 years
  - 1994: Intel Pentium FDIV (floating point division) HW bug ($475 million)
  - 1997: USS Yorktown "smart" warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch ($193 million)

Floating Point Topics

- **Fractional binary numbers**
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

- There are many more details that we won't cover
  - It's a 58-page standard…

Representation of Fractions

- "Binary Point," like decimal point, signifies boundary between integer and fractional parts:

  Example 6-bit representation: **xx.yyyy**

  2^{1} 2^{0} 2^{-1} 2^{-2} 2^{-3} 2^{-4}

- <u>Example</u>: $10.1010_2 = 1{\times}2^{1} + 1{\times}2^{-1} + 1{\times}2^{-3} = 2.625_{10}$

Representation of Fractions

- "Binary Point," like decimal point, signifies boundary between integer and fractional parts:

Example 6-bit representation:

xx.yyyy

2^{1} 2^{0} 2^{-1} 2^{-2} 2^{-3} 2^{-4}

- In this 6-bit representation:
  - What is the encoding and value of the smallest (most negative) number?
  - What is the encoding and value of the largest (most positive) number?
  - What is the smallest number greater than 2 that we can represent?

Scientific Notation (Decimal)

$$\overbrace{6.02}^{\text{mantissa}}{}_{10} \times 10^{23}$$

exponent

decimal point

radix (base)

- *Normalized form*: exactly one digit (non-zero) 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}

Scientific Notation (Binary)

exponent

$$1.\overbrace{01}^{\text{mantissa}}{}_2 \times 2^{-1}$$

binary point

radix (base)

- Computer arithmetic that supports this called floating point due to the "floating" of the binary point
  - Declare such variable in C as `float` (or `double`)

Translating To and From Scientific Notation

- Consider the number $1.011_{two}\times 2^4$
- To convert to ordinary number, shift the decimal to the right by 4
  - Result: $10110_{two} = 22_{ten}$
- For negative exponents, shift decimal to the left
  - $1.011_{two}\times 2^{-2}$ => $0.01011_{two} = 0.34375_{ten}$
- Go from ordinary number to scientific notation by shifting until in *normalized* form
  - 1101.001_{two} => $1.101001_{two}\times 2^3$

"Father" of Floating Point Standard

IEEE Standard 754 for Binary Floating-Point Arithmetic

1989
ACM Turing Award Winner!

Prof. Kahan
Prof. Emeritus
UC Berkeley

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

Floating Point Topics

- Fractional binary numbers
- **IEEE floating-point standard**
- Floating-point operations and rounding
- Floating-point in C

- There are many more details that we won’t cover
  - It’s a 58-page standard…

IEEE Floating Point

- IEEE 754
  - Established in 1985 as uniform standard for floating point arithmetic
  - Main idea: make numerically sensitive programs portable
  - Specifies two things: representation and result of floating operations
  - Now supported by all major CPUs

- Driven by numerical concerns
  - **Scientists**/numerical analysts want them to be as **real** as possible
  - **Engineers** want them to be **easy to implement** and **fast**

    competing goals
  - In the end:
    - Scientists mostly won out
    - Nice standards for rounding, overflow, underflow, but...
    - Hard to make fast in hardware
    - **Float operations can be an order of magnitude slower than integer ops**

      FLOPs

      used in computer benchmarks

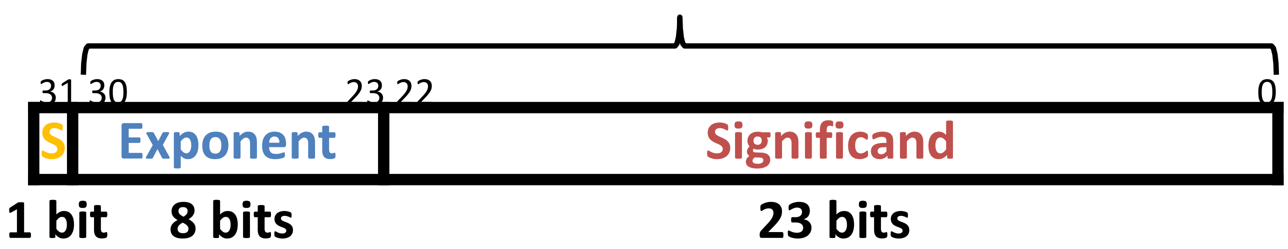
Floating Point Encoding

- Use normalized, base 2 scientific notation:
  - Value: $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$
  - Bit Fields: $(-1)^S \times 1.M \times 2^{(E-\text{bias})}$
- Representation Scheme:
  - Sign bit (0 is positive, 1 is negative)
  - Mantissa (a.k.a. significand) is the fractional part of the number in normalized form and encoded in bit vector **M**
  - Exponent weights the value by a (possibly negative) power of 2 and encoded in the bit vector **E**

1 bit **8 bits** **23 bits**

The Exponent Field

- Why use biased notation for the exponent?
  - Remember that we want floating point numbers to look small when their actual value is small
    - We don't like how in 2's complement, -1 looks bigger than 0. Bias notation preserves the linearity of value



The Exponent Field

- Use biased notation
  - Read exponent as unsigned, but with *bias* of $2^{w-1}-1$ = 127
  - Representable exponents roughly ½ positive and ½ negative
  - Exponent 0 (Exp = 0) is represented as E = 0b 0111 1111
- Why biased?
  - Makes floating point arithmetic easier
  - Makes somewhat compatible with two's complement
- **Practice:** To encode in biased notation, add the bias then encode in unsigned:
  - Exp = 1 → → E = 0b
  - Exp = 127 → → E = 0b
  - Exp = -63 → → E = 0b

The Mantissa (Fraction) Field

31	30 … 23	22 … 0
S	E	M
1 bit	8 bits	23 bits

$$(-1)^{S} \times (1\,.\,M) \times 2^{(E-\text{bias})}$$

- Note the implicit 1 in front of the M bit vector
  - Example: 0b 0011 1111 1100 0000 0000 0000 0000 0000 is read as $1.1_2 = 1.5_{10}$, *not* $0.1_2 = 0.5_{10}$
  - Gives us an extra bit of *precision*
- Mantissa "limits"
  - Low values near M = 0b0…0 are close to 2^{Exp}
  - High values near M = 0b1…1 are close to 2^{Exp+1}

Peer Instruction Question

- What is the correct value encoded by the following floating point number?
  - 0b 0 10000000 11000000000000000000000

A. + 0.75

B. + 1.5

C. + 2.75

D. + 3.5

E. We're lost…

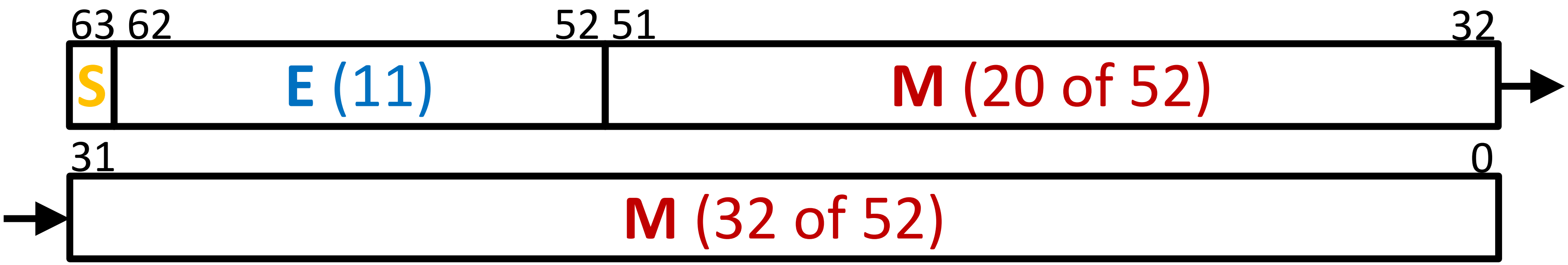
Precision and Accuracy

- Precision is a count of the number of bits in a computer word used to represent a value
  - Capacity for accuracy
- 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 mantissa (highly precise), but is only an approximation (not accurate)

Need Greater Precision?

- Double Precision (vs. Single Precision) in 64 bits



- C variable declared as `double`
- Exponent bias is now $2^{10}–1 = 1023$
- **Advantages:** greater precision (larger mantissa), greater range (larger exponent)
- **Disadvantages:** more bits used, slower to manipulate

Floating Point Numbers Summary

Exponent	Significand	Meaning
0	?	?
0	?	?
1-254	anything	± fl. pt
255	?	?
255	?	?

Representing Zero

- But wait… what happened to zero?
  - Using standard encoding 0x00000000 is $1.0\times2^{-127}\neq0$
    - All because of that dang implicit 1
  - *Special case:* Exp and Significand all zeros = 0
  - Two zeros! But at least 0x00000000 = 0 like integers

	31	30 – 23	22 – 0
+0	0	00000000	00000000000000000000000
	sign	**exponent**	**significand**

	31	30 – 23	22 – 0
-0	1	00000000	00000000000000000000000
	sign	**exponent**	**significand**

Floating Point Numbers Summary

Exponent	Significand	Meaning
0	0	± 0
0	?	?
1-254	anything	± fl. pt
255	?	?
255	?	?

Representing ± ∞

- Division by zero
  - infinity is a number!
  - okay to do further comparison eg. x/0 > y
- Representation
  - Max **exponent** = 255
  - all zero **significand**

	31	30 23	22 0
+ ∞	0	11111111	00000000000000000000000
	sign	exponent	significand

	31	30 23	22 0
- ∞	1	11111111	00000000000000000000000
	sign	exponent	significand

Floating Point Numbers Summary

Exponent	Significand	Meaning
0	0	± 0
0	non-zero	?
1-254	anything	$\pm$ fl. pt
255	0	$\pm \infty$
255	non-zero	?

Representing NaN

- 0/0, sqrt(-4), $\infty - \infty$?
  - Useful for debugging
  - Op(NaN, some number) = NaN
- Representation
  - Max **exponent** = 255
  - non-zero **significand**

+ NaN

31	30 … 23	22 … 0
0	11111111	Non-zero
sign	exponent	significand

- NaN

31	30 … 23	22 … 0
1	11111111	Non-zero
sign	exponent	significand

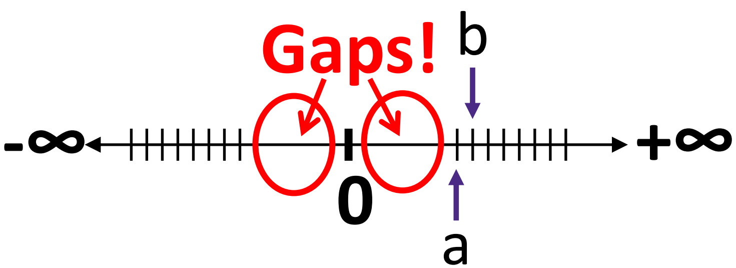
Floating Point Numbers Summary

Exponent	Significand	Meaning
0	0	± 0
0	non-zero	?
1-254	anything	± Norm fl. pt
255	0	± ∞
255	non-zero	NaN

Representing Very Small Numbers

- But wait… what happened to zero?
  - Using standard encoding 0x00000000 =
  - *Special case:* E and M all zeros = 0
    - Two zeros! But at least 0x00000000 = 0 like integers

- New numbers closest to 0:
  - $a = 1.0...0_2 \times 2^{-126} = 2^{-126}$
  - $b = 1.0...01_2 \times 2^{-126} = 2^{-126} + 2^{-149}$
  - Normalization and implicit 1 are to blame
  - *Special case:* E = 0, M ≠ 0 are denormalized numbers

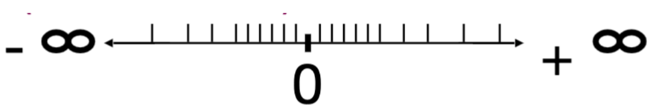


Denorm Numbers

- Short for "denormalized numbers"
  - No leading 1
  - Careful! Implicit exponent = -126 when Exp = 0x00 (intuitive reason: the "binary point" moves one more bit to the left of the leading bit)
- Now what do the gaps look like?

So much closer to 0

  - Smallest denorm: $\pm\ 0.0\ldots01_{two}\times 2^{-126} = \pm\ 2^{-149}$
  - Largest denorm: $\pm\ 0.1\ldots1_{two}\times 2^{-126} = \pm\ (2^{-126} - 2^{-149})$
  - Smallest norm: $\pm\ 1.0\ldots0_{two}\times 2^{-126} = \pm\ 2^{-126}$



No uneven gap! Increments by 2^{-149}

Floating Point Numbers Summary

Exponent	Significand	Meaning
0	0	± 0
0	non-zero	± Denorm fl pt.
1-254	anything	± Norm fl. pt
255	0	± ∞
255	non-zero	NaN

Converting From Hex and Decimal

Convert 0x40600000 to decimal

1 bit for sign, 8 bits for exponent, 23 bits for significand, bias of -127

Step 1: Convert to Binary

0x40600000 = 0100 0000 0110 0000 0000 0000 0000 0000

Step 2: Split Bits Up

0100 0000 0110 0000 0000 0000 0000 0000

Sign	Exponent	Significand
0	100 0000 0	110 0000 0000 0000 0000 0000
1 bit	8 bits	23 bits

Step 3: Check If Norm/Denorm

0100 0000 0110 0000 0000 0000 0000 0000

Sign	Exponent	Significand
0	100 0000 0	110 0000 0000 0000 0000 0000
1 bit	8 bits	23 bits

Exponent is not 00000000, so normalized!

Step 4: Evaluate

$$(-1)^{Sign} * 2^{Exp-Bias} * 1.\mathbf{significand}_2$$

Plug into normalized formula

Step 4: Evaluate

$$(-1)^{Sign} * 2^{Exp-Bias} * 1.\mathbf{significand}_2$$

Plug into normalized formula

0 10000000 11000000000000000000000

Sign = 0, Exp = 128, Bias = 127, 1.significand = 1.11 Ignore trailing 0's

NOTE: In the context of this formula,
Bias = 127

Step 4: Evaluate

$$(-1)^{Sign} * 2^{Exp-Bias} * 1.\mathbf{significand}_2$$

Plug into normalized formula

0 10000000 11000000000000000000000

Sign = 0, Exp = 128, Bias = 127, 1.significand = 1.11 Ignore trailing 0's

$(-1)^{0} * 2^{128 - 127} * 1.11_2 = 2 * 1.11_2$

Step 4: Evaluate

$$(-1)^{Sign} * 2^{Exp-Bias} * 1.\mathbf{significand}_2$$

Plug into normalized formula

0 10000000 11000000000000000000000

Sign = 0, Exp = 128, Bias = -127, 1.significand = 1.11

Ignore trailing 0's

$(-1)^0 * 2^{128-127} * 1.11_2 = 2^1 * 1.11_2$

$= 11.1_2$

exponent is 1 = shifting decimal right by 1

$= 2^1 + 2^0 + 2^{-1}$

$= 3.5$

Converting From Decimal to Binary

Convert -5.625 to binary

1 bit for sign, 8 bits for exponent, 23 bits for significand, bias of -127

Step 1: Convert Left Side of Decimal

-5.625

Ignore sign for now (just make sign bit a 1 at the end)

$5 = 2^2 + 2^0$

$= 101_2$

Step 2: Convert Right Side of Decimal

$.625 = .5 + .125$

$= 2^{-1} + 2^{-3}$

$= .101_2$

Step 3: Combine Both Results and Normalize

$5.625 = 5 + .625$

$= 101_2 + .101_2$

$= 101.101_2$

$101.101_2 = 1.01101_2 * 2^2$

Decimal moved 2 places to the left

Step 4: Convert to Binary

$-1.01101_2 * 2^2$

sign	exponent	significand
1	10000001	01101
(negative)	(2 + 127 for bias)	(ignore implicit 1)

Combine to get: -5.625 = 11000000101101000000000000000000