

CSE 210: Computer Architecture

Lecture 21: Floating Point

Stephen Checkoway
Slides from Cynthia Taylor

Today's Class

- Start floating point

CS History: IEEE 754-1985



William Kahan

Photo credit: George M. Bergman, CC BY-SA 4.0

- Pre-1980, different ISAs used different floating point implementations
- In 1976, John Palmer was managing implementing a floating-point coprocessor at Intel, and wanted a standard floating point
- He went to William Kahan, at UC Berkeley, who worked with Intel to develop a floating point standard
- Kahan, Jerome Coonen and Harold Stone put together a public draft proposal based on Kahan's work with Intel
- This standard was implemented first by Intel in 1980, and then by other manufacturers
- In 1985 it became the official IEEE standard, and stayed the standard until it was updated in 2008

Floating Point

- Problem: Need a way to store non-integer values
- Including numbers with very large and very small magnitudes
- Want to do this the same way for every computer

Base 10

- $123.456 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2} + 6 \times 10^{-3}$
- Digits to the left of the decimal point are multiplied by nonnegative powers of 10
- Digits to the right of the decimal point are multiplied by negative powers of 10

Base 2

- Same thing in base 2 (or any base)
- $110.011 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$
- Binary digits to the left of the binary point are multiplied by nonnegative powers of 2
- Binary digits to the right of the binary point are multiplied by negative powers of 2

$$101.10001_2$$

- 101.10001_2
- Integer part is $101_2 =$
- Fractional part is $0.10001_2 =$
- Total is

Normalized form of numbers in base 10

- Scientific Notation

- 1.2825×10^2
- 2.004×10^{38}
- 3.74×10^{-27}
- -7.888889×10^{40}

- Normalized Form

- Always multiply by power of 10
- Always one nonzero digit before the decimal point

Computers use a normalized version in base 2

- Floating Point Notation
 - $1.11_2 \times 2^2$
 - $1.0101_2 \times 2^{127}$
 - $1.110001_2 \times 2^{-126}$
 - $-1.0001_2 \times 2^{80}$
- Normalized Form
 - One nonzero digit before ~~decimal~~ binary point
 - Multiplied by power of two

We know $101.10001_2 = 5.53125$. What is
 $1.0110001_2 \times 2^2$

A. 1.37578

B. 5.53125

C. 22.0125

D. None of the above

-17.125 in normalized binary

- Step 1. Convert integer part: $17 =$
- Step 2. Convert fractional part: $.125 =$
- Step 3. Add integer and fractional parts: $17.125 =$
- Step 4. Normalize:
- Step 5. Add sign: $-17.125 =$

−0.75 in Binary is

A. $-1.1_2 \times 2^{-1}$

B. $-1.1_2 \times 2^{-2}$

C. $-1.001011_2 \times 2^{-1}$

D. $-1.001011_2 \times 2^{-2}$

E. None of the above

1.2825 * 10² in Binary is

- A. $1.000000001_2 \times 2^{-7}$
- B. $1.000000001_2 \times 2^6$
- C. $1.1001000011001_2 \times 2^6$
- D. $1.000000001_2 \times 2^7$
- E. None of the above

Goal: Represent $(-1)^s * 1.x * 2^e$ in 32 bits

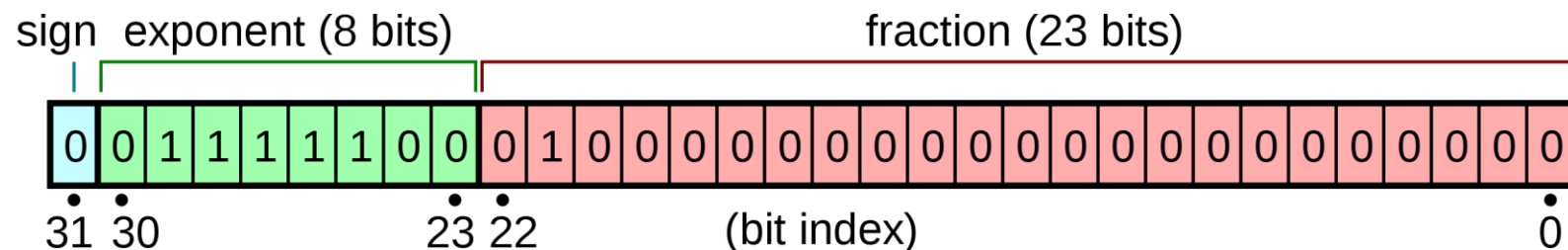
- Divide up 32 bits into different sections
- 1 bit for sign s (1 = negative, 0 = nonnegative)
- 8 bits for exponent e
- 23 bits for significand $1.x$

Goal: Get the most out of 32 bits

- The first number before our ~~decimal~~ binary point is always 1
 - $1.0001 * 2^4$
 - $-1.1011 * 2^{-16}$
- We don't need to represent it in our remaining 23 bits—it is implicit!

$$(-1)^s * 1.\textcolor{red}{x} * 2^e$$

- 1 bit for sign s (1 = negative, 0 = positive)
- 8 bits for exponent e
- 0 bits for implicit leading 1 (called the “hidden bit”)
- 23 bits for significand (without hidden bit)/fraction/mantissa x



$1.001100101 * 2^7$ as a single word

- $1.001100101 * 2^7$ as a single word becomes
 - Sign =
 - Exponent =
 - Fraction =

If we gave more bits to the exponent, and fewer to the fraction, we could represent

- A. Fewer individual numbers
- B. More individual numbers
- C. Numbers with greater magnitude, but less precision
- D. Numbers with smaller magnitude, but greater precision

Want To Make Comparisons Easy

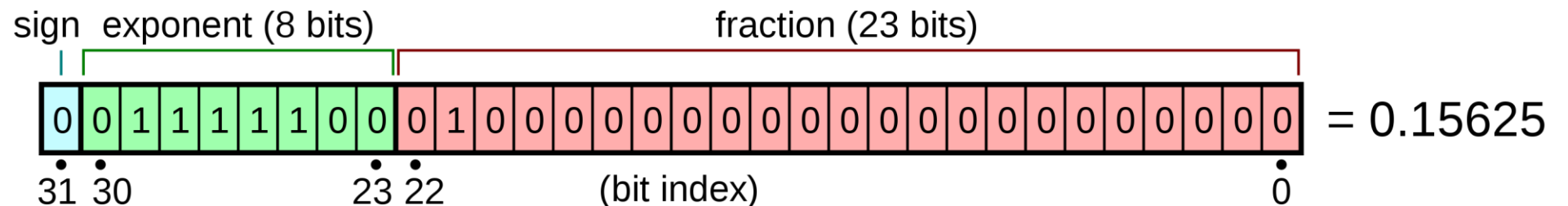
- Can easily tell if number is positive or negative
 - Just check MSB bit
- Exponent is in higher magnitude bits than the fraction
 - Numbers with higher values will look bigger (as integers)
 - 0 00000111 10000000000000000000000000000000 = $1.1 * 2^7$
 - 0 00001000 10000000000000000000000000000000 = $1.1 * 2^8$

Problem with Two's Complement exponents

- 0 00000111 100000000000000000000000000000 = $1.1 * 2^7$
- 0 00001000 100000000000000000000000000000 = $1.1 * 2^8$
- 0 11111000 100000000000000000000000000000 = $1.1 * 2^{-8}$
- Solution: Get rid of negative exponents!
 - We can represent $2^8 = 256$ numbers: normal exponents -126 to 127 and two special values for zero, infinity, (and NaN and subnormals)
 - Add 127 to value of exponent to encode it, subtract 127 to decode

$$(-1)^s * 1.x * 2^e$$

- 1 bit for sign s (1 = negative, 0 = positive)
- 8 bits for exponent e + 127
- 0 bits for implicit leading 1 (called the “hidden bit”)
- 23 bits for significand (without hidden bit)/fraction/mantissa x



Encode 1.000000001×2^7 in 32-bit Floating Point

A. 0 00000111 00000000010000000000000000

B. 0 00000111 10000000010000000000000000

C. 0 10000110 00000000010000000000000000

D. 0 10000110 10000000010000000000000000

E. None of the above

How Can We Represent 0 in Floating Point (as described so far)?

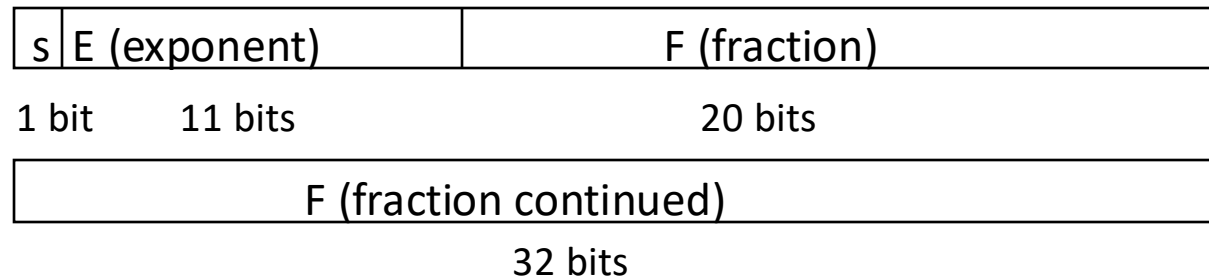
- A. 0 00000000 000000000000000000000000
- B. 0 01111111 000000000000000000000000
- C. 1 00000000 000000000000000000000000
- D. More than one of the above
- E. We can't represent 0

Special Cases

Object	Exponent	Fraction
Zero	0	0
Infinity	255	0
NaN	255	Nonzero

Exception Events in Floating Point

- **Overflow** happens when a positive exponent becomes too large to fit in the exponent field
- **Underflow** happens when a negative exponent becomes too large (in magnitude) to fit in the exponent field
- One way to reduce the chance of underflow or overflow is to offer another format that has a larger exponent field
 - Double precision – takes two MIPS words



Reading

- Next lecture: Floating Point