inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

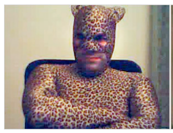
Lecture 15
Floating Point

2010-02-24

Hello to **Robb Neuschwander** listening from **Louisville, KY!**

Lecturer SOE Dan Garcia

www.cs.berkeley.edu/~ddgarcia



Chatroulette site ⇒
Random-pairing videochat, anonymously. Surreal. Watch for NC17 content. ☹

www.nytimes.com/2010/02/21/weekinreview/21bilton.html

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)**
   ⇒ $31{,}556{,}926{,}000_{10}$ (3.1556926_{10} x 10^{10})
2. **Very small numbers? (Bohr radius)**
   ⇒ 0.0000000000529177_{10}m (5.29177_{10} x 10^{-11})
3. **Numbers with <u>both</u> 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 between integer and fractional parts:

Example 6-bit representation: xx.yyyy

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

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

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

0 to 3.9375 (almost 4)

Fractional Powers of 2

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   1.5₁₀
 + 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
    HI    LOW
```

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)

mantissa → $6.02_{10} \times 10^{23}$ ← exponent

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}

Scientific Notation (in Binary)

mantissa → $1.0_{two} \times 2^{-1}$ ← exponent

"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)

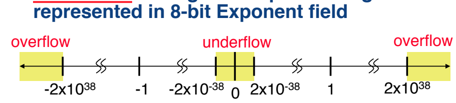
- Normal format: $+1.xxx\ldots x_{two} * 2^{yyy\ldots y_{two}}$
- Multiple of Word Size (32 bits)

31	30 – 23	22 – 0
S	Exponent	Significand
1 bit	8 bits	23 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! ⇒ 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! ⇒ Negative exponent larger than represented in 8-bit Exponent field



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

Double Precision Fl. Pt. Representation

- 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

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`

Administrivia…Midterm in < 2 weeks!

- How should we study for the midterm?
  - Form study groups…don't prepare in isolation!
  - Attend the review session (Time/Location TBA)
  - Look over HW, Labs, Projects, class notes!
  - Go over old exams – HKN office has put them online (link from 61C home page)
  - Attend TA office hours and work out hard probs

IEEE 754 Floating Point Standard (1/3)

Single Precision (DP similar):

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

- 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

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):

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)

"Father" of the Floating point standard

IEEE Standard 754 for Binary Floating-Point Arithmetic.

Prof. Kahan

1989 ACM Turing Award Winner!

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

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:
  - $1 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + \ldots$
  - $= 1+2^{-1}+2^{-3}+2^{-5}+2^{-7}+2^{-9}+2^{-14}+2^{-15}+2^{-17}+2^{-22}$
  - = 1.0 + 0.666115
- Represents: $1.666115_{ten} * 2^{-23} \sim 1.986 * 10^{-7}$ (about 2/10,000,000)

Example: Converting Decimal to FP

-2.340625×10^1

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 = 1.011101101×2^4
5. Convert exponent: 127 + 4 = 10000011_2

1	1000 0011	011 1011 0100 0000 0000 0000

"And in 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 desktop or server computer sold since ~1997 follows these conventions)
- Summary (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)

Understanding the Significand (1/2)

- Method 1 (Fractions):
  - In decimal: 0.340_{10} ⇒ $340_{10}/1000_{10}$ ⇒ $34_{10}/100_{10}$
  - In binary: 0.110_2 ⇒ $110_2/1000_2 = 6_{10}/8_{10}$ ⇒ $11_2/100_2 = 3_{10}/4_{10}$
  - 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: $1.6732 = (1x10^0) + (6x10^{-1}) + (7x10^{-2}) + (3x10^{-3}) + (2x10^{-4})$
  - 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