CS61C – Machine Structures

Lecture 15 - Floating Point Numbers I

2/22/2006

John Wawrzynek

(www.cs.berkeley.edu/~johnw)

www-inst.eecs.berkeley.edu/~cs61c/

Introduction

0011 0100 0101 0101 0100 0011 0100 0010

- What does the above bit pattern mean?

  "4UCB"

  `ori $21, $2, 17218`

  878,003,010

- Data can be anything. Its meaning is determined by how it is interpreted. (As an instruction, as an operand for a arithmetic instruction, …)

- Powerful yet dangerous concept: use integer as pointer, instructions as data, integers as instructions, ...
  (Leads to bugs & security holes in programs)

Review of Number Representation

What numbers can we represent in N bits?

- **Unsigned integers:**

  0 to $2^N - 1$

  (for N=32, 2^N = 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

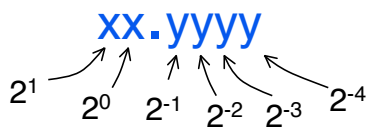
1. **Very large numbers? (seconds/century)**
   $3{,}153{,}384{,}000_{10}$
2. **Very small numbers? (Bohr radius)**
   0.00000000000529177_{10}m
3. **Numbers with both integer and fractional parts?**
   1.5

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 5-bit representation:



$$10.101_2 = 1x2^1 + 1x2^{-1} + 1x2^{-3} = 2.625_{10}$$

If we assume "fixed binary point", range of 5-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? (need left shift by 2)

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 particular binary point.

The binary point can be outside the stored bits, so very large and small numbers can be represented.

Scientific Notation Review

Example:

mantissa → 6.02×10^{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}

Floating Point Representations

Like Scientific Notation for Binary Numbers:

Example:

Mantissa → $1.01_{two} \times 2^{-1}$ ← *exponent*

"binary point" ↑ ↑ *radix (base)*

- In this case, 2 is used for both the mantissa and the exponent part.

Floating Point Representation (1/2)

- Normal format: $+1.xxxxxxxxxx_{two} * 2^{yyyy_{two}}$
- 32-bit version (C "`float`")

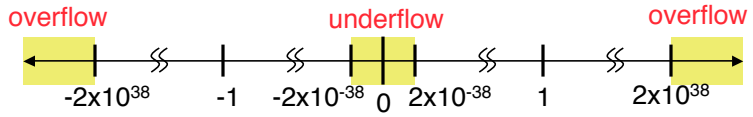
31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

S represents Sign
Exponent represents y's
Significand represents x's

- Represent numbers almost as small as 2.0×10^{-38} to almost 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

° 64 bit version (C "`double`")

31	30 … 20	19 … 0
S	Exponent	Significand (upper bits)
1 bit	11 bits	20 bits

Significand (lower bits)
32 bits

- 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

Administrivia

° Midterm 1, 1 Pimentel, Friday 2/24 6-8pm
- Open Book/Notes, but no electronic devices of any kind!
- Make sure to study machine language concepts and know where to find details in book!
- Other topics for exam …

° How should we study for the midterm?
- Form study groups -- don't prepare in isolation!
- Attend the review session (Wednesday @ 8pm in 10 Evans)
- Look over HW, Labs, Projects
- Review the material you are bringing (textbook, C book)
- Go over old exams – My previous one online. Others available through HKN website?

"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`

IEEE 754 Floating Point Standard (1/4)

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 (2/4)

- IEEE 754 uses "biased exponent" representation.
  - Kahan 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)

IEEE 754 Floating Point Standard (4/4)

- Biased Notation, where bias is number subtracted to get real number
  - IEEE 754 uses bias of 127 for single precision
  - 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

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:
   $10111.01101 = 1.011101101 \times 2^4$
5. Convert exponent: $127 + 4 = 10000011_2$

1	1000 0011	011 1011 0100 0000 0000 0000

Representation for +/- Infinity

- In FP, divide by zero should produce +/- infinity, not overflow.
- Why?
  - OK to do further computations with infinity e.g., X/0 > Y may be a valid comparison
- IEEE 754 represents +/- infinity
  - Largest positive exponent reserved for infinity
  - Significands all zeroes

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	+/- fl. pt. #
255	0	+/- infinity
255	nonzero	???

- Professor Kahan had clever ideas; "Waste not, want not"
  - We'll talk about Exp=0,255 & Sig!=0 later

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).

Things to Remember

- 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 numbers.
- IEEE 754 Floating Point Standard is most widely accepted attempt to standardize interpretation of such numbers
- New MIPS registers(`$f0`-`$f31`), instruct.:
  - Single Precision (32 bits, $2x10^{-38}$… $2x10^{38}$):
    `add.s, sub.s, mul.s, div.s`
  - Double Precision (64 bits , $2x10^{-308}$…$2x10^{308}$):
    `add.d, sub.d, mul.d, div.d`