inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture #11 – Floating Point I

2008-7-9

Upcoming Programming Contests!

QuickTime™ and a TIFF (Uncompressed) decompressor are needed to see this picture.

http://www.icfpcontest.org/ 7/11-7/14

http://code.google.com/codejam/ 7/16 qualifiers

Albert Chae, Instructor

Review

- **MIPS Machine Language Instruction:** 32 bits representing a single instruction

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	target address				

- Branches use PC-relative addressing, Jumps use absolute (actually, pseudo-direct) addressing.
- Disassembly is simple and starts by decoding opcode field. (more tomorrow)

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)
   ⇒ $31{,}556{,}926{,}000_{10}$ ($3.1556926_{10} \times 10^{10}$)
2. Very small numbers? (Bohr radius)
   ⇒ 0.0000000000529177_{10}m ($5.29177_{10} \times 10^{-11}$)
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 (1/2)

- With base 10, we have a decimal point to separate integer and fraction parts to a number.

xx.yyyy

10^1 10^0 10^{-1} 10^{-2} 10^{-3} 10^{-4}

- $20.4005 = 2\times10^1 + 4\times10^{-1} + 5\times10^{-4}$

Representation of Fractions (2/2)

"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}

$10.1010_2 = 1x2^1 + 1x2^{-1} + 1x2^{-3} = 2.625_{10}$

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, exponent, decimal point, radix (base)

$6.02_{10} \times 10^{23}$

- **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, exponent, "binary point", radix (base)

$1.0_{two} \times 2^{-1}$

- **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)

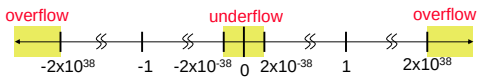
- Normal format: $+1.xxxx\ldots xxx_{two} * 2^{yyy\ldots y_{two}}$
- Multiple of Word Size (32 bits)

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?
  ($> 2.0\times10^{38}$, $< -2.0\times10^{38}$)
  - Overflow! ⇒ Exponent larger than represented in 8-bit Exponent field
- What if result too small?
  (>0 & $< 2.0\times10^{-38}$, <0 & $> -2.0\times10^{-38}$)
  - 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)
- This is currently being worked on
  - The current version has 15 bits for the exponent and 112 bits for the significand
- Oct-Precision?
  - It's been implemented before… (256 bit)
- Half-Precision?
  - Yep, that's for a short (16 bit)
    `en.wikipedia.org/wiki/Quad_precision`
    `en.wikipedia.org/wiki/Half_precision`

Peer Instruction

- Can we represent every number between 0 and 10 using `integers`, `floats`, or neither?
- Gaps between large numbers are smaller because we desire more precision when talking about large quantities. True or False
- There is no interval of numbers where the gap between a floating point representation and the next biggest is 1. True or False

Administrivia

- Assignments
  - Proj1 due 7/11 @ 11:59pm (preliminary ag)
  - Quiz 5/6 due 7/14 @ 11:59pm
  - HW3 due 7/15 @ 11:59pm
  - Proj2 due 7/18 @ 11:59pm

Administrivia…Midterm in < 2 weeks!

- Midterm Mon 2008-07-21@7-10pm, 155 Dwinelle
  - Conflicts/DSP? Email me
  - You can bring green sheet and one handwritten double sided note sheet
- How should we study for the midterm?
  - Form study groups…don't prepare in isolation!
  - Attend the faux midterm/review session
  - Look over HW, Labs, Projects, class notes!
  - Write up your handwritten 1-page study sheet
  - 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 (1/5)

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 (2/5)

- Negative Exponent?
  - 2's comp? 1.0 x 2^{-1} v. 1.0 x2^{+1} (1/2 v. 2)

1/2	0	1111 1111	000 0000 0000 0000 0000 0000
2	0	0000 0001	000 0000 0000 0000 0000 0000

  - This notation using integer compare of 1/2 v. 2 makes 1/2 > 2!

IEEE 754 Floating Point Standard (3/5)

- Designers wanted FP numbers to be used even if no FP hardware; e.g., sort records with FP numbers using integer compares
- Possible solutions?
  - Could break FP number into 3 parts:
    - compare signs, then compare exponents, then compare significands
  - Wanted it to be faster, single compare (using integer hardware) if possible
- For a faster (integer) compare:
  - Highest order bit is sign (negative < positive)
  - Exponent next, so bigger exponent => bigger #
  - Significand last: exponents same, bigger significand => bigger #

IEEE 754 Floating Point Standard (4/5)

- Instead, pick notation so that 0000 0001 is most negative, and 1111 1111 is most positive
  - 1.0 x 2^{-1} v. 1.0 x2^{+1} (1/2 v. 2)

1/2	0	0111 1110	000 0000 0000 0000 0000 0000
2	0	1000 0000	000 0000 0000 0000 0000 0000

- To accomplish this, IEEE 754 uses "biased exponent" representation.

IEEE 754 Floating Point Standard (5/5)

- 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

- $(-1)^S$ x (1 + Significand) x $2^{(Exponent-127)}$
  - Bias is usually $2^{(\#expbits - 1)}$ – 1 (why?)

"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)

Converting Decimal to FP (1/4)

- Simple Case: If denominator is an exponent of 2 (2, 4, 8, 16, etc.), then it's easy.
- Show MIPS representation of -0.75
  - -0.75 = -3/4
  - $-11_{two}/100_{two} = -0.11_{two}$
  - Normalized to $-1.1_{two} \times 2^{-1}$
  - $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
  - $(-1)^1 \times (1 + .100\ 0000 \ldots 0000) \times 2^{(126-127)}$

1	0111 1110	100 0000 0000 0000 0000 0000

Converting Decimal to FP (2/4)

- Not So Simple Case: If denominator is not an exponent of 2.
  - Then we can't represent number precisely, but that's why we have so many bits in significand: for precision
  - Once we have significand, normalizing a number to get the exponent is easy.
  - So how do we get the significand of a neverending rational number?

Converting Decimal to FP (3/4)

- Fact: All rational numbers have a repeating pattern when written out in decimal.
- Fact: This still applies in binary.
- To finish conversion:
  - Write out binary number with repeating pattern.
  - Cut it off after correct number of bits (different for single v. double precision).
  - Derive Sign, Exponent and Significand fields.

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

Converting Decimal to FP (4/4)

-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

Understanding the Significand (1/2)

- Method 1 (Fractions):
  - In decimal: 0.340_{10} $\Rightarrow 340_{10}/1000_{10}$ $\Rightarrow 34_{10}/100_{10}$
  - In binary: 0.110_2 $\Rightarrow 110_2/1000_2 = 6_{10}/8_{10}$ $\Rightarrow 11_2/100_2 = 3_{10}/4_{10}$
  - 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: $1.6732 = (1\times10^0) + (6\times10^{-1}) + (7\times10^{-2}) + (3\times10^{-3}) + (2\times10^{-4})$
  - In binary: $1.1001 = (1\times2^0) + (1\times2^{-1}) + (0\times2^{-2}) + (0\times2^{-3}) + (1\times2^{-4})$
  - 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

Peer Instruction

1	1000 0001	111 0000 0000 0000 0000 0000

What is the decimal equivalent of the floating pt # above?

```
1: -3.5
2: -7
3: -7.5
4: -7 * 2^129
5: -129 * 2^7
```


Peer Instruction Answer

What is the decimal equivalent of:

1	1000 0001	111 0000 0000 0000 0000 0000
S	Exponent	Significand

$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

$(-1)^1 \times (1 + .111) \times 2^{(129-127)}$

$-1 \times (1.111) \times 2^{(2)}$

-111.1

-7.5_{ten}

```
1: -3.5
2: -7
3: -7.5
4: -7 * 2^129
5: -129 * 2^7
```


"And in conclusion…"

- Floating Point numbers <u>approximate</u> values that we want to use.
- 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, bias of 1023