Instructor
Paul Pearce

inst.eecs.berkeley.edu/~cs61c

UCB CS61C : Machine Structures

Lecture 11
Floating Point

2010-07-08

DID YOU KNOW? HELLA IS A SCIENTIFIC TERM

Well, not quite yet. A UC Davis student (shown left) is attempting to standardize "hella" as an SI prefix meaning 10^{27}. Google calculator already recognizes it! Coming soon to CS61C Lecture 1?

http://tinyurl.com/2b7rvth

In 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 addressing.
- Disassembly is simple and starts by decoding **opcode** field. (more Monday)

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

"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
```

```
  01.100   1.5₁₀
  00.100   0.5₁₀
  -------
  00 000
 000 00
0110 0
00000
00000
----------
0000110000
```

Multiplication a bit more complex:

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…

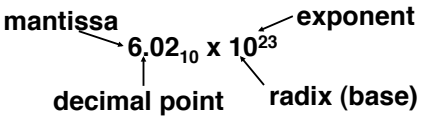
Store these bits and keep track of the binary point 2 places to the left of the MSB

Any other solution would lose accuracy!

With floating point rep., each numeral carries a exponent field recording the whereabouts of its binary point.

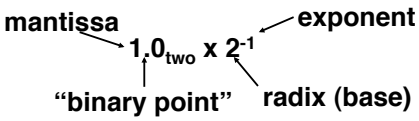
The binary point can be outside the stored bits, so very large and small numbers can be represented.

Scientific Notation (in Decimal)



- 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}

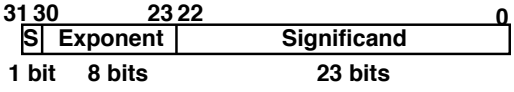
Scientific Notation (in Binary)



- 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)

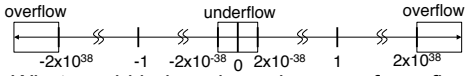
- Normalized format: $+1.xxx\ldots x_{two} * 2^{yyy\ldots y_{two}}$
- Multiple of Word Size (32 bits)



- **S represents Sign**
  Exponent represents y's
  Significand represents x's
- **Represent numbers as small as ~2.0 x 10^{-38} to as large as ~2.0 x 10^{38}**

Floating Point Representation (2/2)

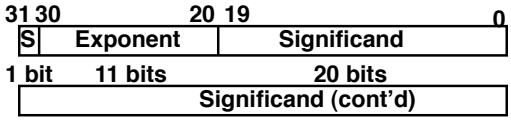
- What if result too large?
  (> 2.0×10^{38} , < -2.0×10^{38})
  - Overflow! ⇒ Exponent larger than represented in 8-bit Exponent field
- What if result too small?
  (>0 & < 2.0×10^{-38} , <0 & > -2.0×10^{-38})
  - Underflow! ⇒ Negative exponent larger than represented in 8-bit Exponent field



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

Double Precision Fl. Pt. Representation

- Next Multiple of Word Size (64 bits)



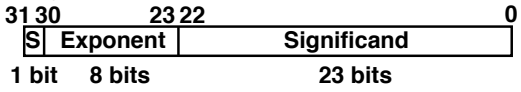
- **Double Precision (vs. Single Precision)**
  - **C variable declared as `double`**
  - **Represent numbers almost as small as 2.0×10^{-308} to almost as large as 2.0×10^{308}**
  - **But primary advantage is greater accuracy due to larger significand**

Administrivia…

- How should we study for the midterm?
  - Form study groups…don't prepare in isolation!
  - Attend the review session (Monday 6pm Location TBA)
  - Look over HW, Labs, Projects, class notes!
  - Go over old exams – HKN has put them online (link from 61C home page)
  - Attend TA office hours, my office hours, work out hard probs
  - ASK QUESTIONS!
- Additional jump example on Monday.

IEEE 754 Floating Point Standard (1/3)

Single Precision (DP similar):

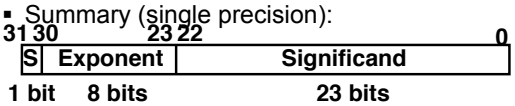


- Sign bit: 1 means negative
  0 means positive
- Significand:
  - To pack more bits, leading 1 implicit for normalized numbers
  - always true: 0 < Significand < 1 (for normalized numbers)
  - Written as: 1.Significand
  - Book uses: 1 + Significand Same thing!
- Note: 0 has no leading 1, so reserve exponent value 0 just for number 0

IEEE 754 Floating Point Standard (2/3)

- **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):



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

- **Double precision identical, except with exponent bias of 1023 (half, quad similar)**

IEEE 754 Floating Point Standard (3/3)

- 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)
  - With this implementation 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

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: $1.6732 = (1x10^0) + (6x10^{-1}) + (7x10^{-2}) + (3x10^{-3}) + (2x10^{-4})$
  - In binary: $1.1001 = (1x2^0) + (1x2^{-1}) + (0x2^{-2}) + (0x2^{-3}) + (1x2^{-4})$
  - Each digit has a specific value that is a power of 2 if to the left of the binary point, and a fractional power of 2 if to the right.
  - Advantage: good for quickly calculating significand value; use this method for translating FP numbers

"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

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

- Sign: 0 ➔ positive
- Exponent:
  - $0110\ 1000_{two} = 104_{ten}$
  - Bias adjustment: 104 - 127 = -23
- Significand:
  - 1 . 10101010100001101000010
  - $1 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + \ldots$
  - $= 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22}$
  - $= 1.0 + 0.666115$

- **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:
   $23 = 16 + 4 + 2 + 1 = 10111_2$
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 ± ∞

- In FP, divide by 0 should produce ± ∞, not overflow.
- Why?
  - OK to do further computations with ∞ E.g., X/0 > Y may be a valid comparison
  - Ask math majors
- IEEE 754 represents ± ∞
  - Most positive exponent reserved for ∞
  - 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	+/- ∞
255	nonzero	???

- Professor Kahan had clever ideas; "Waste not, want not"
  - What about Exp=0,255 & Sig!=0

Representation for Not a Number

- What do I get if I calculate `sqrt(-4.0)` or `0/0`?
  - If ∞ not an error, these shouldn't be either
  - Called Not a Number (NaN)
  - Exponent = 255, Significand nonzero
- Why is this useful?
  - Hope NaNs help with debugging?
  - They contaminate: op(NaN, X) = NaN

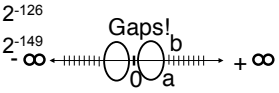
Representation for Denorms (1/2)

- Problem: There's a gap among representable FP numbers around 0
  - Smallest representable pos num:
    $a = 1.000\ldots0_2 * 2^{-126} = 2^{-126}$
  - Second smallest representable pos num:
    $b = 1.000\ldots\ldots1_2 * 2^{-126}$
    $= (1 + 0.00\ldots1_2) * 2^{-126}$
    $= (1 + 2^{-23}) * 2^{-126}$
    $= 2^{-126} + 2^{-149}$

Normalization and implicit 1 is to blame!

$a - 0 = 2^{-126}$

$b - a = 2^{-149}$

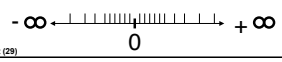


Representation for Denorms (2/2)

- Solution:
  - We still haven't used Exponent=0, Significand nonzero
  - Denormalized number: no (implied) leading 1, implicit exponent = -126

$$(-1)^S \times (0 . \text{Significand}) \times 2^{(-126)}$$

  - Smallest representable pos num:
    - $A = 2^{-149}$
  - Second smallest representable pos num:
    - $b = 2^{-148}$



Special Numbers Summary

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	+/-0
0	nonzero	Denorm
1-254	Anything	+/- fl. Pt #
255	0	+/- ∞
255	nonzero	NaN

Peer Instruction

- Let `f(1,2)` = # of floats between 1 and 2
- Let `f(2,3)` = # of floats between 2 and 3

A: f(1,2) < f(2,3)
B: f(1,2) = f(2,3)
C: f(1,2) > f(2,3)

Peer Instruction

What is the decimal equivalent of:

1	1000 0001	111 0000 0000 0000 0000 0000

```
a) -7 * 2^129
b) -3.5
c) -3.75
d) -7
e) -7.5
```

"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)

"Also in conclusion…"

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	+/-0
0	nonzero	Denorm
1-254	Anything	+/- fl. Pt #
255	0	+/- ∞
255	nonzero	NaN

Bonus slides

- These are extra slides that used to be included in lecture notes, but have been moved to this, the "bonus" area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

Bonus

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).

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

Rounding

- When we perform math on real numbers, we have to worry about rounding to fit the result in the significant field.
- The FP hardware carries two extra bits of precision, and then round to get the proper value
- Rounding also occurs when converting:
  - double to a single precision value, or
  - floating point number to an integer

IEEE FP Rounding Modes

- Halfway between two floating point values (rounding bits read 10)? Choose from the following:
- Round towards + ∞
  - Round “up”: 1.01 10 → 1.10 , -1.01 10 → -1.01
- Round towards - ∞
  - Round “down”: 1.01 10 → 1.01, -1.01 10 → -1.10
- Truncate
  - Just drop the extra bits (round towards 0)
- Unbiased (default mode). Round to nearest EVEN number
  - Half the time we round up on tie, the other half time we round down. Tends to balance out inaccuracies.
  - In binary, even means least significant bit is 0.
- Otherwise, not halfway (00, 01, 11)! Just round to the nearest float.

FP Addition

- More difficult than with integers
- Can’t just add significands
- How do we do it?
  - De-normalize to match exponents
  - Add significands to get resulting one
  - Keep the same exponent
  - Normalize (possibly changing exponent)
- Note: If signs differ, just perform a subtract instead.

MIPS Floating Point Architecture (1/4)

- MIPS has special instructions for floating point operations:
  - Single Precision:
    `add.s, sub.s, mul.s, div.s`
  - Double Precision:
    `add.d, sub.d, mul.d, div.d`
- These instructions are far more complicated than their integer counterparts. They require special hardware and usually they can take much longer to compute.

MIPS Floating Point Architecture (2/4)

- Problems:
  - It’s inefficient to have different instructions take vastly differing amounts of time.
  - Generally, a particular piece of data will not change from FP to int, or vice versa, within a program. So only one type of instruction will be used on it.
  - Some programs do no floating point calculations
  - It takes lots of hardware relative to integers to do Floating Point fast

MIPS Floating Point Architecture (3/4)

- 1990 Solution: Make a completely separate chip that handles only FP.
- Coprocessor 1: FP chip
  - contains 32 32-bit registers: `$f0`, `$f1`, …
  - most registers specified in `.s` and `.d` instruction refer to this set
  - separate load and store: `lwc1` and `swc1` ("load word coprocessor 1", "store …")
  - Double Precision: by convention, even/odd pair contain one DP FP number: `$f0`/`$f1`, `$f2`/`$f3`, … , `$f30`/`$f31`

MIPS Floating Point Architecture (4/4)

- 1990 Computer actually contains multiple separate chips:
  - Processor: handles all the normal stuff
  - Coprocessor 1: handles FP and only FP;
  - more coprocessors?… Yes, later
  - Today, cheap chips may leave out FP HW
- Instructions to move data between main processor and coprocessors:
  - `mfc0`, `mtc0`, `mfc1`, `mtc1`, etc.
- Appendix pages A-70 to A-74 contain many, many more FP operations.

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

Casting floats to ints and vice versa

`(int)` *`floating_point_expression`*

Coerces and converts it to the nearest integer (C uses truncation)

```
i = (int) (3.14159 * f);
```

`(float)` *`integer_expression`*

converts integer to nearest floating point

```
f = f + (float) i;
```


int → float → int

```
if (i == (int)((float) i)) {
 printf("true");
}
```

- Will not always print "true"
- Most large values of integers don't have exact floating point representations!
- What about `double`?

float → int → float

```
if (f == (float)((int) f)) {
 printf("true");
}
```

- Will not always print "true"
- Small floating point numbers (<1) don't have integer representations
- For other numbers, rounding errors

Floating Point Fallacy

- FP add associative: FALSE!
  - x = – 1.5 x 10^{38}, y = 1.5 x 10^{38}, and z = 1.0
  - x + (y + z) = –1.5x10^{38} + (1.5x10^{38} + 1.0)
    = –1.5x10^{38} + (1.5x10^{38}) = 0.0
  - (x + y) + z = (–1.5x10^{38} + 1.5x10^{38}) + 1.0
    = (0.0) + 1.0 = 1.0
- Therefore, Floating Point add is not associative!
  - Why? FP result approximates real result!
  - This example: 1.5 x 10^{38} is so much larger than 1.0 that 1.5 x 10^{38} + 1.0 in floating point representation is still 1.5 x 10^{38}

Peer Instruction

1. **Converting `float` -> `int` -> `float` produces same `float` number**
2. **Converting `int` -> `float` -> `int` produces same `int` number**
3. **FP add is associative:**
   `(x+y)+z = x+(y+z)`

	123
a)	FFF
b)	FFT
c)	FTF
c)	FTT
d)	TFF
d)	TFT
e)	TTF
e)	TTT

Peer Instruction Answer

1. **Converting `float` -> `int` -> `float` produces same `float` number** FALSE
2. **Converting `int` -> `float` -> `int` produces same `int` number** FALSE
3. **FP add is associative:**
   `(x+y)+z = x+(y+z` FALSE

1. 3.14 -> 3 -> 3
2. 32 bits for signed int, but 24 for FP mantissa?
3. x = biggest pos #, y = -x, z = 1 (x != inf)

	123
a)	FFF
b)	FFT
c)	FTF
c)	FTT
d)	TFF
d)	TFT
e)	TTF
e)	TTT