inst.eecs.berkeley.edu/~cs61c

UC Berkeley CS61C : Machine Structures

Lecture 16
Floating Point II

2010-02-26

TA Michael Greenbaum
www.cs.berkeley.edu/~cs61c-tf

"Research without Google would be like life without electricity"

"A *Nature News* survey of Chinese scientists found that 84 percent of them thought losing access to Google would "somewhat or significantly" hurt their work process. Like their American counterparts, Chinese researchers use Google and Google Scholar to find papers and related information.

http://www.wired.com/wiredscience/2010/02/china-scientists-google/

Review

Exponent tells Significand how much (2^i) to count by (…, 1/4, 1/2, 1, 2, …)

- 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)

"Father" of the Floating point standard

IEEE Standard 754 for Binary Floating-Point Arithmetic.

1989 ACM Turing Award Winner!

Prof. Kahan

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

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).

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)

ExponentSignificand		Object
0	0	0
0	nonzero	???
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	???

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

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

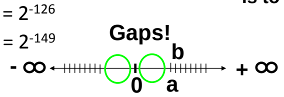
Representation for Denorms (1/2)

- Problem: There's a gap among representable FP numbers around 0
  - Smallest representable pos num:
    $a = 1.0\ldots_2 * 2^{-126} = 2^{-126}$
  - Second smallest representable pos num:
    $b = 1.000\ldots\ldots1_2 * 2^{-126}$
    $= (1 + 0.00\ldots1_2) * 2^{-126}$
    $= (1 + 2^{-23}) * 2^{-126}$
    $= 2^{-126} + 2^{-149}$

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

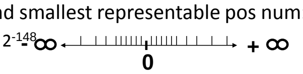
  $b - a = 2^{-149}$

Normalization and implicit 1 is to blame!



Representation for Denorm (2/2)

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

Rounding

- When we perform math on real numbers, we have to worry about rounding to fit the result in the significand field.
- The FP hardware carries two extra bits of precision, and then rounds 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.

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)`

```
   ABC
1: FFF
2: FFT
3: FTF
4: FTT
5: TFF
```

Peer Instruction

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

```
1: f(1,2) < f(2,3)
2: f(1,2) = f(2,3)
3: f(1,2) > f(2,3)
```

"And in conclusion…"

- Reserve exponents, significands:

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

- 4 Rounding modes (default: unbiased)
- MIPS Fl ops complicated, expensive

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

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