CS61C – Machine Structures

Lecture 16 - Floating Point Numbers II

2/24/2006

John Wawrzynek

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

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

IEEE 754 Floating Point Standard (review)

- **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).

Administrivia

- Midterm 1, 1 Pimentel, Tonight 6-8pm sharp
  - Open Book/Notes, but no electronic devices of any kind!
- Don't forget to work on homework and start project 3 over the weekend.

Representation for Not a Number

- What do I get if I calculate `sqrt(-4.0)` or `0/0`?
  - If infinity is 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

Special Numbers (cont'd)

° 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	NaN

Representation for Denorms (1/2)

° Problem: There's a gap among representable FP numbers around 0

- Smallest representable pos num:

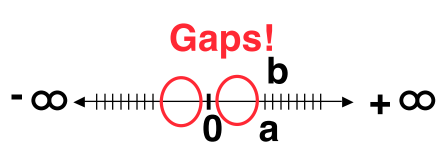
  $a = 1.0\ldots_2 * 2^{-126} = 2^{-126}$

- Second smallest representable pos num:

  $b = 1.000\ldots\ldots1_2 * 2^{-126} = 2^{-126} + 2^{-149}$

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

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



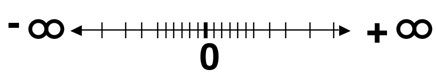
Representation for Denorms (2/2)

° Solution:

- We still haven't used Exponent = 0, Significand nonzero
- Denormalized number: no (implied) leading 1, exponent = -126.
- Smallest representable pos num:

  a = 2^{-149}
- Second smallest representable pos num:

  b = 2^{-148}



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

- Round towards +infinity
  - ALWAYS round “up”: 2.001 → 3

    -2.001 → -2
- Round towards -infinity
  - ALWAYS round “down”: 1.999 → 1,

    -1.999 → -2
- Truncate
  - Just drop the last bits (round towards 0)
- Round to (nearest) even
  - Normal rounding, almost

Round to Even

- Round like you learned in grade school
- Except if the value is right on the borderline, in which case we round to the nearest EVEN number

  2.5 → 2

  3.5 → 4
- Insures fairness on calculation
  - This way, half the time we round up on tie, the other half time we round down
  - Tends to balance out inaccuracies

This is the default rounding mode

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) 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}

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 so 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.**

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`