inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture #11 – Floating Point II

2007-7-12

Scott Beamer, Instructor

Sony & Nintendo make E3 News

AP / Stefano Paltera

www.nytimes.com

Review

- 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):

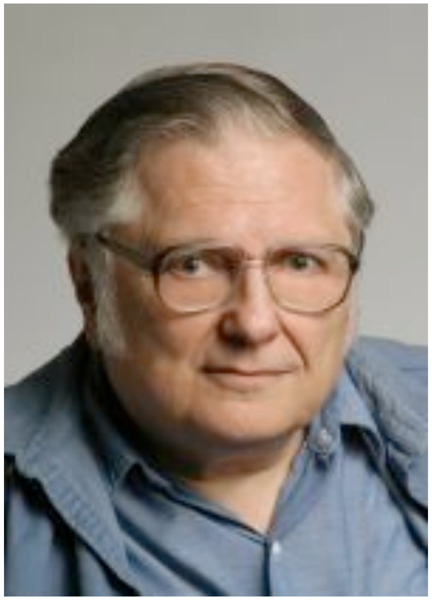
31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
  - Double precision identical, bias of 1023

"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).

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

"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}

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

If we assume "fixed binary point", range of 6-bit representations with this format:

0 to 3.9375 (almost 4)

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 = $(1 \times 10^{0}) + (6 \times 10^{-1}) + (7 \times 10^{-2}) + (3 \times 10^{-3}) + (2 \times 10^{-4})$
  - In binary: 1.1001 = $(1 \times 2^{0}) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$
  - 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

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:
  - $1 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + \ldots$
    $= 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22}$
    $= 1.0_{ten} + 0.666115_{ten}$
- Represents: $1.666115_{ten} * 2^{-23} \sim 1.986 * 10^{-7}$

  (about 2/10,000,000)

Converting Decimal to FP (1/3)

- 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 (2/3)

- 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 number?

Converting Decimal to FP (3/3)

- 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

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 too
  - What about sign?
    - `+0: 0 00000000 00000000000000000000000`
    - `-0: 1 00000000 00000000000000000000000`
- Why two zeroes?
  - Helps in some limit comparisons
  - Ask math majors

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"
  - Exp=0,255 & Sig!=0 …

Representation for Not a Number

- What is `sqrt(-4.0)` or `0/0`?
  - If ∞ not an error, these shouldn't be either.
  - Called <u>N</u>ot <u>a</u> <u>N</u>umber (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:

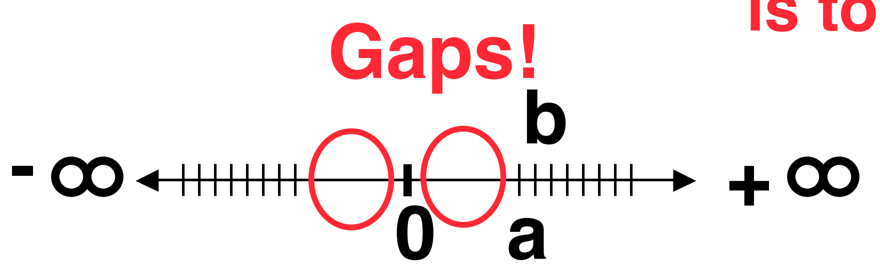
    $a = 1.0\ldots_2 * 2^{-126} = 2^{-126}$
  - Second smallest representable pos num:

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

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

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

Normalization and implicit 1 is to blame!

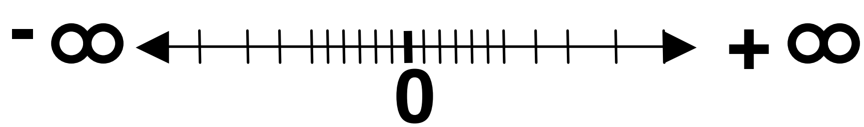


Representation for Denorms (2/2)

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

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

    $b = 2^{-148}$



Overview

- Reserve exponents, significands:

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

Peer Instruction

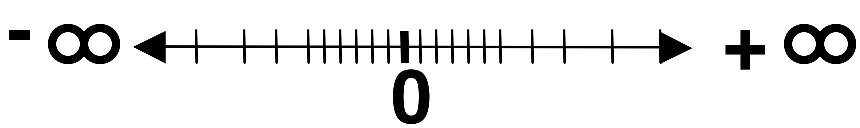
- Let `f(1,2)` = # of floats between 1 and 2
- Let `f(2,3)` = # of floats between 2 and 3

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

Peer Instruction Answer

- 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)
```



Administrivia…Midterm in 11 days!

- Midterm 10 Evans Mon 2007-7-23 @ 7-10pm
  - Conflicts/DSP? Email Me
- How should we study for the midterm?
  - Form study groups -- don't prepare in isolation!
  - Attend the review session (2007-7-20 - Time & Place TBD)
  - Look over HW, Labs, Projects
  - Write up your 1-page study sheet--handwritten
  - Go over old exams – HKN office has put them online (link from 61C home page)
- If you have trouble remembering whether it's +127 or –127
  - remember the exponent bits are <u>unsigned</u> and have max=255, min=0, so what do we have to do?

More Administrivia

- **Assignments**
  - **Proj1 due tonight @ 11:59pm**
  - **HW4 due 7/15 @ 11:59pm**
  - **Proj2 posted, due 7/20 @ 11:59pm**
- **Mistaken Green Sheet Error**
  - **When I said jal was R[31] = PC + 8, it should be R[31] = PC + 4 until after the midterm**
  - **Treat it this way on HW4 and Midterm**

Rounding

- Math on real numbers ⇒ we worry about rounding to fit result in the significant field.
- FP hardware carries 2 extra bits of precision, and rounds for proper value
- Rounding occurs when…
  - converting double to single precision
  - converting floating point # to an integer
  - Intermediate step if necessary

IEEE Four Rounding Modes

- Round towards + ∞
  - ALWAYS round "up": 2.1 ⇒ 3, -2.1 ⇒ -2
- Round towards - ∞
  - ALWAYS round "down": 1.9 ⇒ 1, -1.9 ⇒ -2
- Round towards 0 (I.e., truncate)
  - Just drop the last bits
- Round to (nearest) even (default)
  - Normal rounding, almost: 2.5 ⇒ 2, 3.5 ⇒ 4
  - Like you learned in grade school (almost)
  - Insures fairness on calculation
  - Half the time we round up, other half down
  - Also called Unbiased

Integer Multiplication (1/3)

- **Paper and pencil example (unsigned):**

```
Multiplicand  1000      8
Multiplier   x1001      9
              1000
             0000
            0000
         +1000
          01001000
```

  - m bits x n bits = m + n bit product

Integer Multiplication (2/3)

- In MIPS, we multiply registers, so:
  - 32-bit value x 32-bit value = 64-bit value
- Syntax of Multiplication (signed):
  - `mult` register1, register2
  - Multiplies 32-bit values in those registers & puts 64-bit product in special result regs:
    - puts product upper half in hi, lower half in lo
  - hi and lo are 2 registers separate from the 32 general purpose registers
  - Use `mfhi` register & `mflo` register to move from hi, lo to another register

Integer Multiplication (3/3)

- **Example:**
  - **in C:** `a = b * c;`
  - **in MIPS:**
    - let b be $s2; let c be $s3; and let a be $s0 and $s1 (since it may be up to 64 bits)

```
mult $s2,$s3   # b*c
mfhi $s0       # upper half of
               # product into $s0
mflo $s1       # lower half of
               # product into $s1
```

- **Note: Often, we only care about the lower half of the product.**

Integer Division (1/2)

- **Paper and pencil example (unsigned):**

```
                1001      Quotient
Divisor 1000|1001010     Dividend
           -1000
               10
               101
               1010
              -1000
                  10     Remainder
             (or Modulo result)
```

- **Dividend = Quotient x Divisor + Remainder**

Integer Division (2/2)

- **Syntax of Division (signed):**
  - `div` register1, register2
  - Divides 32-bit register 1 by 32-bit register 2:
  - puts remainder of division in `hi`, quotient in `lo`
- **Implements C division (`/`) and modulo (`%`)**
- **Example in C:**

```
a = c / d;
b = c % d;
```

- **in MIPS:** `a↔$s0;b↔$s1;c↔$s2;d↔$s3`

```
div  $s2,$s3     # lo=c/d, hi=c%d
mflo $s0         # get quotient
mfhi $s1         # get remainder
```

Unsigned Instructions & Overflow

- MIPS also has versions of `mult`, `div` for unsigned operands:

```
multu

divu
```

  - Determines whether or not the product and quotient are changed if the operands are signed or unsigned.

- MIPS <u>does not</u> check overflow on ANY signed/unsigned multiply, divide instr
  - Up to the software to check `hi`

FP Addition & Subtraction

- Much more difficult than with integers (can't just add significands)
- How do we do it?
  - De-normalize to match larger exponent
  - Add significands to get resulting one
  - Normalize (& check for under/overflow)
  - Round if needed (may need to renormalize)
- If signs ≠, do a subtract. (Subtract similar)
  - If signs ≠ for add (or = for sub), what's ans sign?
- Question: How do we integrate this into the integer arithmetic unit? [Answer: We don't!]

MIPS Floating Point Architecture (1/4)

- **Separate floating point instructions:**
  - **Single Precision:**
    `add.s, sub.s, mul.s, div.s`
  - **Double Precision:**
    `add.d, sub.d, mul.d, div.d`
- **These are <u>far more complicated</u> than their integer counterparts**
  - **Can take much longer to execute**

MIPS Floating Point Architecture (2/4)

- Problems:
  - Inefficient to have different instructions take vastly differing amounts of time.
  - Generally, a <u>particular piece of data will not change FP ⇔ `int`</u> within a program.
    - Only 1 type of instruction will be used on it.
  - Some programs <u>do no FP calculations</u>
  - It takes lots of hardware relative to integers to do FP 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 of the 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`
    - Even register is the name

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, FP coprocessor integrated with CPU, or cheap chips may leave out FP HW
- Instructions to move data between main processor and coprocessors:
    - `mfc0`, `mtc0`, `mfc1`, `mtc1`, etc.
- Appendix contains many more FP ops

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
6:	TFT
7:	TTF
8:	TTT

Peer Instruction Answer

"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

- Integer `mult`, `div` uses `hi`, `lo` regs
    - `mfhi` and `mflo` copies out.
- Four rounding modes (to even default)
- MIPS FL ops complicated, expensive