CS61C
Floating Point Operations & Multiply/Divide

Lecture 9

February 17, 1999

Dave Patterson
(http.cs.berkeley.edu/~patterson)

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

Review 1/2

- <u>**Big Idea: Instructions determine meaning of data; nothing inherent inside the data**</u>
- **Characters: ASCII takes one byte**
  - MIPS support for characters: `lbu`, `sb`
- **C strings: Null terminated array of bytes**
- **Floating Point Data: approximate representation of very large or very small numbers in 32-bits or 64-bits**
  - IEEE 754 Floating Point Standard
  - Driven by Berkeley's Professor Kahan

Review 2/2: Floating Point Representation

- **Single Precision and Double Precision**

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

31	30 … 20	19 … 0
S	Exponent	Significand
1 bit	11 bits	20 bits

Significand (cont'd)
32 bits

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

Outline

- Fl. Pt. Representation, a little slower
- Floating Point Add/Sub
- MIPS Floating Point Support
- Kahan crams more in 754: Nan, ∞
- Administrivia, "What's this stuff Good for"
- Multiply, Divide
- Example: C to Asm for Floating Point
- Conclusion

Floating Point Basics

- Fundamentally 3 fields to represent Floating Point Number
  - Normalized Fraction (Mantissa/Significand)
  - Sign of Fraction
  - Exponent
  - Represents $(-1)^S \times (\text{Fraction}) \times 2^{\text{Exponent}}$ where 1 <= Fraction < 2 (i.e., normalized)
- If number bits left-to-right s1, s2, s3, ... then represents number

$$(-1)^S x(1+(s1x2^{-1})+(s2x2^{-2})+(s3x2^{-3})+...)x\ 2^{\text{Exponent}}$$

Order 3 Fields in a Word?

- "Natural": Sign, Fraction, Exponent?
  - Problem: If want to sort using integer operations, won't work:
  - 1.0×2^{20} vs. 1.1×2^{10} ; latter looks bigger!

0	10000	10100

0	11000	01010

- Exponent, Sign, Fraction?
  - Need to get sign first, since negative < positive
- Therefore order is

Sign	Exponent	Fraction

How Stuff More Precision into Fraction?

- In normalized form, so fraction is either:

1.xxx xxxx xxxx xxxx xxxx xxx
or
0.000 0000 0000 0000 0000 000

- Trick: If hardware automatically places 1 in front of binary point of normalized numbers, then get 1 more bit for the fraction, increasing accuracy "for free"

1.xxx xxxx xxxx xxxx xxxx xxx
becomes
(1).xxx xxxx xxxx xxxx xxxx xxx<u>x</u>

  - Comparison OK; "subtracting" 1 from both

How differentiate from Zero in Trick Format?

- 1.0000 ... 000 => . 0000 ... 000<u>0</u>
- Solution: Reserve most negative exponent to be only used for Zero; rest are normalized so prepend a 1
- Convention is

0	-Big	00000	=> 0.00000
0	> -Big	00000	=> $1.00000 \times 2^{\text{Exp}}$

Negative Exponent?

- 2's comp? 1.0×2^{-1} v. $1.0 \times 2^{+1}$ (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 look greater than 2!
- Instead, pick notation 0000 0000 is most negative, 1111 1111 is most positive
  - Called <u>Biased Notation</u>; <u>bias</u> subtracted to get number
  - 127 in Single Prec. (1023 D.P.)
  - Zero is 0 0000 0000 0...0 0000

-127	0000 0000
-126	0000 0001
	...
-1	0111 1110
0	0111 1111
+1	1000 0000
	...
+127	1111 1110
+128	1111 1111

Example: Converting Fl. Pt. to Decimal

0	0110 1000	101 0101 0100 0011 0100 0010

- $(-1)^S \times (1+\text{Significand}) \times 2^{(\text{Exponent-Bias})}$
- Sign: 0 => $(-1)^0 = 1$ => positive
- Exponent: $0110\ 1000_{two} = 104_{ten}$
  - Bias adjustment: 104 - 127 = -13
  - Represents 2^{-13}
- Fraction:
  - Exponent not most negative (!= 0000 0000) so prepend a 1
  - <u>1</u>.101 0101 0100 0011 0100 0010

Example: Converting Fl. Pt. to Decimal

0	0110 1000	101 0101 0100 0011 0100 0010

- Significand: $1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + \ldots$
  - $1+2^{-1}+2^{-3}+2^{-5}+2^{-7}+2^{-9}+2^{-14}+2^{-15}+2^{-17}+2^{-22}$

    = 1+ 1/2 + 1/8 + 1/32 +1/128 + 1/512 + 1/16384 + 1/32768 + 1/131072 + 1/4194304
  - Mutiply fractions by 4194304 (GCD) for sum

    = 1.0 + (2097152 + 524288 + 131072 + 32768 + 8192 + 256 + 128 + 32 + 1)/4194304
    = 1.0 + (2793889)/4194304
    = 1.0 + 0.66612
- Bits represent: $+1.66612_{ten} * 2^{-13} \sim +2.034*10^{-4}$

Basic Fl. Pt. Addition Algorithm

For addition (or subtraction) of X to Y (X<Y):

(1) Compute $D = Exp_Y - Exp_X$ (align binary point)

(2) Right shift $(1+Sig_X)$ D bits=>$(1+Sig_X)*2^{(ExpX-ExpY)}$

(3) Compute $(1+Sig_X)*2^{(ExpX - ExpY)} + (1+Sig_Y)$

Normalize if necessary; continue until MS bit is 1
(4) Too small (e.g., 0.001xx...)
left shift result, decrement result exponent
(4') Too big (e.g., 101.1xx…)
right shift result, increment result exponent

(5) If result significand is 0, set exponent to 0

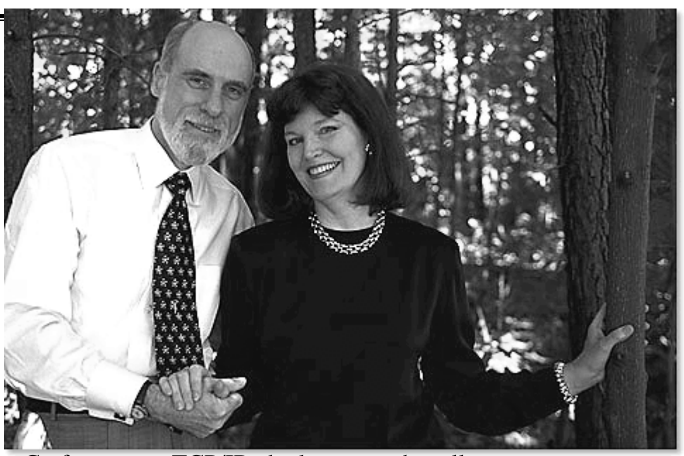
MIPS Floating Point Architecture

- Single Precision, Double Precision versions of add, subtract, multiply, divide, compare
  - Single `add.s`, `sub.s`, `mul.s`, `div.s`, `c.lt.s`
  - Double `add.d`, `sub.d`, `mul.d`, `div.d`, `c.lt.d`
- Registers?
  - Simplest solution: use existing registers
  - Normally integer and Fl.Pt. ops on different data, for performance could have separate registers
  - MIPS adds 32 32-bit Fl. Pt. reg: `$f0`, `$f1`, `$f2` ...,
  - Thus need Fl. Pt. data transfers: `lwc1`, `swc1`
  - Double Precision? Even-odd pair of registers (`$f0#$f1`) act as 64-bit register: `$f0`, `$f2`, `$f4`, ...

Administrivia

- Readings: 4.8 (skip HW), 3.9
- 5th homework: Due 2/24 7PM
  - Exercises 4.21, 4.25, 4.28
- 3rd Project/5th Lab: MIPS Simulator Due Wed. 3/3 7PM
- Midterm conflict time: Mon 3/15 6-9PM
- Backup for lecture notes in case main page unavailable:
  - `www.cs.berkeley.edu/~pattrsn/61CS99`

"What's This Stuff Good For?"



In 1974 Vint Cerf co-wrote TCP/IP, the language that allows computers to communicate with one another. His wife of 35 years (Sigrid), hearing-impaired since childhood, began using the Internet in the early 1990s to research cochlear implants, electronic devices that work with the ear's own physiology to enable hearing. Unlike hearing aids, which amplify all sounds equally, cochlear implants allow users to clearly distinguish voices--even to converse on the phone. Thanks in part to information she gleaned from a chat room called "Beyond Hearing," Sigrid decided to go ahead with the implants in 1996. The moment she came out of the operation, she immediately called home from the doctor's office--a phone conversation that Vint still relates with tears in his eyes. ***One Digital Day*, 1998 (www.intel.com/onedigitalday)**

Special IEEE 754 Symbols: Infinity

- Overflow is not same as divide by zero
- IEEE 754 represents +/- infinity
  - OK to do further computations with infinity e.g., X/0 > Y may be a valid comparison
  - Most positive exponent reserved for infinity
  - Try printing 1.0/0.0 and see what is printed

Greedy Kahan: what else can I put in?

- What defined so far ? (Single Precision)

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	+/- fl. pt. number
255	0	+/- infinity
255	nonzero	???

- Represent Not a Number; e.q. sqrt(-4); called NaN
  - Exp. = 255, Significand nonzero
  - They contaminate: op(NaN,X) = NaN
  - Hope NaNs help with debugging?
  - Only valid operations are ==, !=

Greedy Kahan: what else can I put in?

- What defined so far (Single Precision)?

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	
255	0	+/- infinity
255	nonzero	NaN

- Exp. = 0, Significand nonzero?
  - Can we get greater precision?
- Represent very, very small numbers (> 0, < smallest normalized number); Denomarlized Numbers (COD p. 300)
  - Ignore denorms for CS61C

MULTIPLY (unsigned): Terms, Example

- Paper and pencil example (unsigned):

```
Multiplicand  1000
Multiplier    1001
              1000
             0000
            0000
           1000
Product   01001000
```

  - m bits x n bits = m+n bit product
- MIPS: `mul`, `mulu` puts product in pair of new registers `hi`, `lo`; copy by `mfhi`, `mflo`
  - 32-bit integer result in `lo`; Logically overflow if product too big, but software must check `hi`

Multiply by Power of 2 via Shift Left

- Number representation: $d_{31}d_{30} \ldots d_2d_1d_0$
  - $d_{31} \times 2^{31} + d_{30} \times 2^{30} + \ldots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$
- What if multiply by 2?
  - $d_{31} x 2^{31+1} + d_{30} x 2^{30+1} + \ldots + d_2 x 2^{2+1} + d_1 x 2^{1+1} + d_0 x 2^{0+1}$ $= d_{31} x 2^{32} + d_{30} x\ 2^{31} + \ldots + d_2 \times 2^3 + d_1 \times 2^2 + d_0 \times 2^1$
- What if shift left by 1?
  - $d_{31} \times 2^{31} + d_{30} \times 2^{30} + \ldots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0$ $\Rightarrow d_{30} x 2^{31} + d_{29} x\ 2^{30} + \ldots + d_2 x 2^3 + d_1 x 2^2 + d_0 x 2^1$
- Multiply by 2^i often replaced by shift left i
  - Compiler usually does this; try it yourself

Divide: Terms, Review Paper & Pencil

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

Dividend = Quotient x Divisor + Remainder

- **MIPS: `div`, `divu` puts Remainer into `hi`, puts Quotient into `lo`**

Example with Fl Pt, Multiply, Divide?

```
void mm (double x[][], double y[][],
                 double z[][]){
 int i, j, k;

 for (i=0; i!=32; i=i+1)
   for (j=0; j!=32; j=j+1)
     for (k=0; k!=32; k=k+1)
         x[i][j] = x[i][j] + y[i][k]
                       * z[k][j];

}
```

- **Starting addresses are parameters in \$a0, \$a1, and \$a2. Integer variables are in \$t3, \$t4, \$t5. Arrays 32 by 32**
- **Use pseudoinstructions: `li` (load immediate), `l.d`/`s.d` (load/store 64 bits)**

Floating Point Fallacies: Add Associativity?

- **$x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, and $z = 1.0$**
- **$x + (y + z) = -1.5\text{x}10^{38} + (1.5\text{x}10^{38} + 1.0)$**

  $= -1.5\text{x}10^{38} + (1.5\text{x}10^{38}) = \underline{0.0}$
- **$(x + y) + z = (-1.5\text{x}10^{38} + 1.5\text{x}10^{38}) + 1.0$**

  $= (0.0) + 1.0 = \underline{1.0}$
- **<u>Therefore, Floating Point add not associative!</u>**
  - **1.5×10^{38} is so much larger than 1.0 that $1.5 \times 10^{38} + 1.0$ is still 1.5×10^{38}**
  - **Fl. Pt. result approximation of real result!**

Shift Right Arithmetic; Divide by 2???

- **Shifting right by n bits would seem to be the same as dividing by 2n**
- **Problem is signed integers**
  - **Zero fill is wrong for negative numbers**
- **Shift Right Arithmetic (`sra`); sign extends (replicates sign bit); does it work?**
- **Divide -5 by 4 via `sra` 2; result should be -1**
- **1111 1111 1111 1111 1111 1111 1111 1011**
- **1111 1111 1111 1111 1111 1111 1111 1110**
- **= -2, not -1; Off by 1, so doesn't work**

Floating Point Fallacy: Accuracy optional?

- July 1994: Intel discovers bug in Pentium
  - Occasionally affects bits 12-52 of D.P. divide
- Sept: Math Prof. discovers, put on WWW
- Nov: Front page trade paper, then NYTimes
  - Intel: "several dozen people that this would affect. So far, we've only heard from one."
  - Intel claims customers see 1 error/27000 years
  - IBM claims 1 error/month, stops shipping
- Dec: Intel apologizes, replace chips $300M
- Reputation? What responsibility to society?

New MIPS arithmetic instructions

Example	Meaning	Comments
mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
multu$2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
div $2,$3	Lo = $2 ÷ $3,	Lo = quotient, Hi = rem
divu $2,$3	Lo = $2 ÷ $3,	Unsigned quotient, rem.
mfhi $1	$1 = Hi	Used to get copy of Hi
mflo $1	$1 = Lo	Used to get copy of Lo
add.s $0,$1,$2	$f0=$f1+$f2	Fl. Pt. Add (single)
add.d $0,$2,$4	$f0=$f2+$f4	Fl. Pt. Add (double)
sub.s $0,$1,$2	$f0=$f1-$f2	Fl. Pt. Subtract (single)
sub.d $0,$2,$4	$f0=$f2-$f4	Fl. Pt. Subtract (double)
mul.s $0,$1,$2	$f0=$f1x$f2	Fl. Pt. Multiply (single)
mul.d $0,$2,$4	$f0=$f2x$f4	Fl. Pt. Multiply (double)
div.s $0,$1,$2	$f0=$f1÷$f2	Fl. Pt. Divide (single)
div.d $0,$2,$4	$f0=$f2÷$f4	Fl. Pt. Divide (double)
c.X.s $0,$1	flag1= $f0 X $f1	Fl. Pt.Compare (single)
c.X.d $0,$2	flag1= $f0 X $f2	Fl. Pt.Compare (double)

X is eq, lt, le; bc1t, bc1f tests flag

"And in Conclusion.." 1/1

- IEEE 754 Floating Point standard: accuracy first class citizen
- Computer numbers have limited size => limited precision
  - underflow: too small for Fl. Pt. (bigger negative exponent than can represent)
  - overflow: too big for Fl. Pt. or integer (bigger positive exponent than can represent, or bigger integer than fits in word)
  - Programmers beware!
- Next: Starting a program