CS61C - Machine Structures

Lecture 9 - Floating Point, Part I

September 27, 2000

David Patterson

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

Overview

- Floating Point Numbers
- Motivation: Decimal Scientific Notation
  - Binary Scientific Notatioin
- Floating Point Representation inside computer (binary)
  - Greater range, precision
- Decimal to Floating Point conversion, and vice versa
- Big Idea: Type is not associated with data
- MIPS floating point instructions, registers

Review of Numbers

- Computers are made to deal with numbers
- What can we represent in N bits?
  - Unsigned integers:

    0 to $2^N - 1$
  - Signed Integers (Two's Complement)

    $-2^{(N-1)}$ to $2^{(N-1)} - 1$

Other Numbers

- What about other numbers?
  - Very large numbers? (seconds/century)
    $3{,}155{,}760{,}000_{10}$ ($3.15576_{10} \times 10^9$)
  - Very small numbers? (atomic diameter)
    0.00000001_{10} ($1.0_{10} \times 10^{-8}$)
  - Rationals (repeating pattern)
    2/3 (0.666666666. . .)
  - Irrationals
    $2^{1/2}$ (1.414213562373. . .)
  - Transcendentals
    e (2.718...), π (3.141...)
- All represented in scientific notation

Scientific Notation Review

mantissa → 6.02×10^{23} ← *exponent*

decimal point ↑ *radix (base)*

- 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 for Binary Numbers

Mantissa → $1.0_{two} \times 2^{-1}$ ← *exponent*

"binary point" ↑ *radix (base)*

- Computer arithmetic that supports it called floating point, because it represents numbers where binary point is not fixed, as it is for integers
  - Declare such variable in C as `float`

Floating Point Representation (1/2)

° Normal format: $+1.\text{xxxxxxxxxx}_{two} * 2^{\text{yyyy}_{two}}$

° Multiple of Word Size (32 bits)

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

° S represents Sign
Exponent represents y's
Significand represents x's

° Represent numbers as small as 2.0×10^{-38} to as large as 2.0×10^{38}

Floating Point Representation (2/2)

° What if result too large? (> $2.0\text{x}10^{38}$)

- Overflow!
- Overflow => Exponent larger than represented in 8-bit Exponent field

° What if result too small? (>0, < $2.0\text{x}10^{-38}$)

- Underflow!
- Underflow => Negative exponent larger than represented in 8-bit Exponent field

° How to reduce chances of overflow or underflow?

Double Precision Fl. Pt. Representation

° Next Multiple of Word Size (64 bits)

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

Significand (cont'd)
32 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

IEEE 754 Floating Point Standard (1/4)

° Single Precision, DP similar

° Sign bit: 1 means negative
0 means positive

° Significand:

- To pack more bits, leading 1 implicit for normalized numbers
- 1 + 23 bits single, 1 + 52 bits double
- always true: 0 < Significand < 1 (for normalized numbers)

° Note: 0 has no leading 1, so reserve exponent value 0 just for number 0

IEEE 754 Floating Point Standard (2/4)

° Kahan wanted FP numbers to be used even if no FP hardware; e.g., sort records with FP numbers using integer compares

° Could break FP number into 3 parts: compare signs, then compare exponents, then compare significands

° Wanted it to be faster, single compare if possible, especially if positive numbers

° Then want order:

- Highest order bit is sign (negative < positive)
- Exponent next, so big exponent => bigger #
- Significand last: exponents same => bigger #

IEEE 754 Floating Point Standard (3/4)

° Negative Exponent?

- 2's comp? 1.0×2^{-1} v. $1.0 \text{x}2^{+1}$ (1/2 v. 2)

	S	Exponent	Significand
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 > 2!

° Instead, pick notation 0000 0001 is most negative, and 1111 1111 is most positive

- 1.0×2^{-1} v. $1.0 \text{x}2^{+1}$ (1/2 v. 2)

	S	Exponent	Significand
1/2	0	0111 1110	000 0000 0000 0000 0000 0000
2	0	1000 0000	000 0000 0000 0000 0000 0000

IEEE 754 Floating Point Standard (4/4)

- Called **Biased Notation**, where bias is number subtract 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):

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

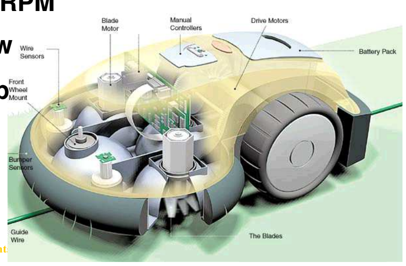
Administrivia

- Need to catchup with Homework
- Reading assignment: Reading 4.8

What's this stuff good for? Mow Lawn?

- Robot lawn mower: "Robomow RL-500"
- Surround lawn, trees with perimeter wire
- Sense tall grass to spin blades faster: up to 5800 RPM
- 71 lbs. Slow if senses object, stop if bumps
- $795

See N.Y.Times, May 18, 2000 "Pull Up a Lawn Chair and Watch the Robot Mow the Grass"



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:
  - $1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + \ldots$
    $= 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22}$
    $= 1.0 + 0.666115$
- Represents: $1.666115_{ten} \times 2^{-23} \sim 1.986 \times 10^{-7}$ (about 2/10,000,000)

Continuing Example: Binary to ???

0011 0100 0101 0101 0100 0011 0100 0010

- Convert 2's Comp. Binary to Integer:

  $2^{29}+2^{28}+2^{26}+2^{22}+2^{20}+2^{18}+2^{16}+2^{14}+2^{9}+2^{8}+2^{6}+2^{1}$
  $= 878{,}003{,}010_{ten}$
- Convert Binary to Instruction:

0011 01	00 010	1 0101	0100 0011 0100 0010
13	2	21	17218

```
ori $s5, $v0, 17218
```

- Convert Binary to ASCII:

0011 0100	0101 0101	0100 0011	0100 0010
4	U	C	B

Big Idea: Type not associated with Data

0011 0100 0101 0101 0100 0011 0100 0010

- What does bit pattern mean:
  - 1.986 $*10^{-7}$? 878,003,010? "4UCB"? `ori $s5, $v0, 17218`?
- Data can be anything; operation of instruction that accesses operand determines its type!
  - Side-effect of stored program concept: instructions stored as numbers
- Power/danger of unrestricted addresses/ pointers: use ASCII as Fl. Pt., instructions as data, integers as instructions, ... (Leads to security holes in programs)

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.

Hairy Example (1/2)

- How to represent 1/3 in MIPS?
- 1/3

  $= 0.33333\ldots_{10}$

  $= 0.25 + 0.0625 + 0.015625 + 0.00390625 + 0.0009765625 + \ldots$

  $= 1/4 + 1/16 + 1/64 + 1/256 + 1/1024 + \ldots$

  $= 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + 2^{-10} + \ldots$

  $= 0.0101010101\ldots_2 * 2^0$

  $= 1.0101010101\ldots_2 * 2^{-2}$

Hairy Example (2/2)

- Sign: 0
- Exponent = -2 + 127 = 125_{10}=01111101_2
- Significand = 0101010101…

0	0111 1101	0101 0101 0101 0101 0101 010

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
  - Ask math majors
- IEEE 754 represents +/- infinity
  - Most positive exponent reserved for infinity
  - 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	+/- infinity
255	nonzero	???

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

FP Addition

- Much 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.

FP Subtraction

- Similar to addition
- How do we do it?
  - De-normalize to match exponents
  - Subtract significands
  - Keep the same exponent
  - Normalize (possibly changing exponent)

FP Addition/Subtraction

° Problems in implementing FP add/sub:
  • If signs differ for add (or same for sub), what will be the sign of the result?

° 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 instructions are far more complicated than their integer counterparts, so they can take much longer.

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 numbers *approximate* values that we want to use.

° IEEE 754 Floating Point Standard is most widely accepted attempt to standardize interpretation of such numbers ($1T)

° New MIPS registers(`$f0`-`$f31`), instruct.:
  • Single Precision (32 bits, $2\text{x}10^{-38}$… $2\text{x}10^{38}$):
    `add.s, sub.s, mul.s, div.s`
  • Double Precision (64 bits , $2\text{x}10^{-308}$…$2\text{x}10^{308}$):
    `add.d, sub.d, mul.d, div.d`

° Type is not associated with data, bits have no meaning unless given in context