ECE 271 – Microcomputer Architecture and Applications Lecture 21

Vince Weaver

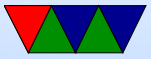
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

9 April 2019

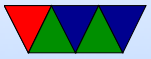
Announcements

- Read Chapter 12



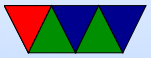
IEEE 754 Floating Point

- Standard from 1985



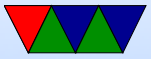
Floating Point Layout

- $(-1)^{sign}(1.fraction) \times 2^{exponent-bias}$
- exponent=0 means zero (or subnormal). What is all 0 value?
- exponent=bias NaN, +/- infinity



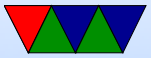
Floating Point Sizes

- Half precision (16 bit)
  Sign=1, Exponent=5, fraction=10
- Single precision (32 bit) (float in C)
  Sign=1, Exponent=8, Fraction=23
- Double precision (64 bit) (double in C)
  Sign=1, Exponent=11, Fraction=52
- Intel x86 (80 bits)
- Why have smaller sizes? They take up less room and are faster.

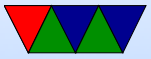


Floating Point Conversion Examples

- You have the value `0xc1ff0000`. If it's a 32-bit floating point value, what is the decimal equivalent?
- `1100 0001 1111 1111 0000 0000 0000 0000`
- Sign bit is 1 (negative)
- Exponent is 8 bits, `1000 0011` which is 131
- Fraction is `1111 1110 0000 0000 0000 000`
  so 0.1111 111 $1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128 = 0.9921875$
- $f = (-1)^S \times (1 + fraction) \times 2^{exponent-127}$

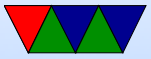


- $f = (-1)^{-1} \times (1 + 0.9921875) \times 2^{131-127}$
- $f = -1 \times 1.9921875 \times 2^{4}$
- $f = -31.875$



Convert FP to binary

- You want to convert the value 6.022×10^{23}
- Sign bit is 0 (positive)
- 6.022 in binary is:
  110 (six)
  .022 .5? 0
  .25 0
  .125 0
  .0625 0
  .03125 0



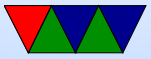
.015625 1 R .006375

.0078125 0

.00390625 1 R .00246875

.001953125 1 R .00515625

- So 110.000001011. . .
- Wait, 10**23? What's *that* in base 2? ln (6.022E23) / ln(2) = 78.994589
  subtract 78, then $2^{.994589} = 1.9925128$
  So it equals $1.9925128 * 2^{78}$
  1.
  .5? 1 R 0.4925128



.25? 1 R 0.245128

.125? 1 R 0.1175128

.0625? 1 R 0.0550128

.03125? 1 R 0.0237628

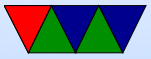
.015625? 1 R 0.0081378

.0078125? 1 R 0.0003253

.00390625? 0 R 0.0003253

.001953125? 0 So 1.11111110

- In 32-bit floating point
  - $f = (-1)^{-1} \times (1 + 11111100) \times 2^{78}$



X-127 = 78, X = 205 = 1100 1101

0 110,0 110,1 111,1111,0 000,0000,0 000,0000

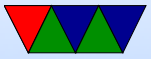
0x66ff0000

- In 64-bit floating point 1/11/52

  X-1023=78, X=1101 = 10001001101

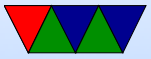
  0 100,0100,1101, 1111 1110 0000 0000 0000 0000 0000 0000 0000 ...

  0x44dfe000 00000000



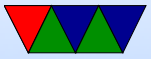
Floating Point Conversion Examples

```
int main(int argc, char **argv) {

        float f;
        unsigned int x;

        x=0x66ff0000;
        memcpy(&f,&x,sizeof(float));
        printf("%x␣is␣%g\n",x,f);
        f=6.022e23;
        memcpy(&x,&f,sizeof(float));
        printf("%x␣is␣%g\n",x,f);
        return 0;
}
66ff0000 is 6.02102e+23
66ff0aa8 is 6.022e+23
44dfe00000000000 is 6.02102e+23
44dfe154f457ea13 is 6.022e+23
```

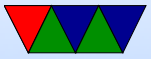


Special Values

- Zero ... cannot represent in standard form (why? because it has to be 1.x in the mantissa). Special value, all zeros. Positive and Negative zeros.
- Positive and negative infinity: all exponent bits are 1s, mantissa all zeros
- NaN (not a number).
  Exponent all 1s, mantissa non-zero
  Things like 0/0, sqrt(-1), log(-1). Two types: QNaN (quiet) which does not cause an exception, and SNaN

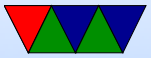


(signaling) that cause an exception that needs o be handled.



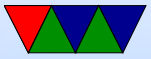
Overflow and Underflow

- What's the smallest number you can represent? Smallest exponent 0b00000001, fraction 0000, so $(-1)^S \times (1+0) \times 2^{1-127} = \pm 1.18 \times 10^{-38}$
- What's the largest number?
  Maximum exponent 0b11111110 and mantissa all 1s
  $(-1)^S \times (1+1-2^{-23}) \times 2^{254-127} = \pm 3.40 \times 10^{38}$
- What is underflow? Too small but not zero?
- Overflow, too large to be represented



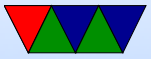
Subnormal Numbers

- Numbers between smallest and zero
- If exponent is 0, treat leading digit in mantissa as 0 instead of 1
- Can get down to 1.45×10^{-45}



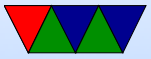
Tradeoff between range and resolution

-



Floating Point Comparison

- `float f = (5.0 - 1.0/7.0) + (1.0/7.0);`
- `if (f==5.0) printf("Five␣exactly.\n")`
- May work may not. Best way is to have some error (epsilon) and compare if absolute value less than a number.

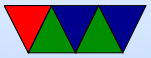


Floating Point Rounding Rules

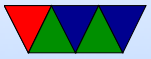
- Complex mess, can cause interesting issues
- Especially as in floating point there are extra bits usually kept around for accuracy, but they are rounded off when written out to memory and you have to fit exactly in 32 or 64 bits
- IEEE-754
  - Round to nearest

    What do we do if *exactly* in between?

    round to even

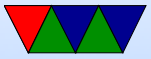


- Round toward zero (truncate)
- Round to +inf (round up)
- Round toward -inf (round down)



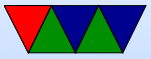
Floating Point Addition

- Shift smaller fraction to match larger one
- add or subtract based on sign bits
- normalize the sum
- round to appropriate bits
- detect overflow and underflow
- do example



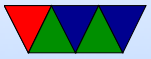
Floating Point Multiplication

- Identify the sign
- add the exponents
- multiply the fractions (including leading hidden one)
- normalize the results

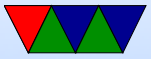


Transcendentals?

- cordic?

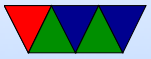


Quake Square Root Trick



Floating Point Drawbacks

- special hardware
- power hungry, if not commonly used
- chip area, expense
- back in day, special chip
- rounding issues
- money calcs. 1/10 only approximate. .0001100110011



- trouble near zero