CSCI 2021: Binary Floating Point Numbers

Chris Kauffman

Last Updated:
Mon Feb 18 11:12:55 CST 2019

Logistics

Reading Bryant/O'Hallaron

- Ch 2.4-5 (Floats, now)
- Ch 3.1-7 (Assembly Intro, soon)
- 2021 Quick Guide to GBD

Goals

- Finish Bitwise ops
- gdb introduction
- Floating Point layout
- A2 Overview (Wednesday)

Lab04

- Bit operations, floats, gdb
- New grading policy to fill up half-empty labs

  30% Check-off during lab
  20% Check-off outside lab

Assignment 2

- Problem 1: Bit shift operations (50%)
- Problem 2: Puzzlebox via debugger (50% + makeup)

Parts of a Fractional Number

The meaning of the "decimal point" is as follows:

$$\begin{aligned} 123.406_{10} = & 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + \\ & 4 \times 10^{-1} + 0 \times 10^{-2} + 6 \times 10^{-3} \\ = & 123.406_{10} \end{aligned} \qquad \begin{aligned} 123 &= 100 + 20 + 3 \\ 0.406 &= \frac{4}{10} + \frac{6}{1000} \end{aligned}$$

Changing to base 2 induces a "binary point" with similar meaning:

$$\begin{aligned} 110.101_2 = & 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + \\ & 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ = & 6.625_{10} \end{aligned} \qquad \begin{aligned} 6 &= 4 + 2 \\ 0.625 &= \frac{1}{2} + \frac{1}{8} \end{aligned}$$

Thus fractional numbers can be represented in binary as well assuming conventions

- X bits for integer part before binary point
- Y bits for fractional part after binary point

Scientific notation

Should be familiar with "scientific" or "engineering" notation for numbers with a fractional part

Standard	Scientific	`printf("%.4e",x);`
123.456	1.23456×10^{2}	`1.2346e+02`
50.01	5.001×10^{1}	`5.0010e+01`
3.14159	3.14159×10^{0}	`3.1416e+00`
0.54321	5.4321×10^{-1}	`5.4321e-01`
0.00789	7.89×10^{-3}	`7.8900e-03`

- **Always** includes one digit prior to decimal place
- Has some **significant** digits after the decimal place
- Multiplies by a **power of 10** to get actual number

Binary Floating Point Layout Uses Scientific Convention

- Some bits for integer/fractional part
- Some bits for exponent part
- All in base 2: 1's and 0's, powers of 2

Conversion Example

Below steps convert a decimal number to a fractional binary number equivalent then adjusts to scientific representation.

```
float fl = -248.75;


              7  6  5  4 3 2 1 0   -1  -2
-248.75 = -(128+64+32+16+8+0+0+0).(1/2+1/4)
        = -11111000.11 *2^0
           76543210 12
        = -1111100.011 *2^1
           6543210 123
        = -111110.0011 *2^2
           543210 1234
           ...
             MANTISSA    EXPONENT
        = -1.111100011 * 2^7
           0 123456789
```

Mantissa $\equiv$ Signifcand $\equiv$ Fractional Part

IEEE 754 Format: *The* Standard for Floating Point

`float`	`double`	Property
32	64	Total bits
1	1	Bits for sign (1 neg / 0 pos)
8	11	Bits for Exponent multiplier (power of 2)
23	52	Bits for Fractional part or **mantissa**
7.22	15.95	Decimal digits of accuracy[1]

- IEEE floating point standard is the most commonly implemented version of format and hardware to do arithmetic
- Numbers appear in several forms

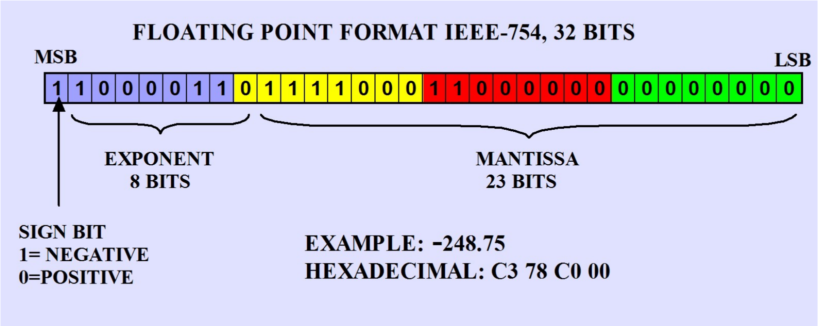
  Normalized most common

  Denormalized close to zero, exponent zero

  Special extreme/error values, exponent maxed

[1] Wikipedia: IEEE 754

Example `float` Layout of -248.75: `float_bits.c`



Source: IEEE-754 Tutorial, www.puntoflotante.net

Color: 8-bit blocks, **Negative**: highest bit, leading 1

Exponent: high 8 bits, 2^7 encoded with bias of -127

```
  1000_0110 - 0111_1111
= 128+4+2   - 127
= 134 - 127
= 7
```

Fractional/Mantissa portion is

```
1.111100011
^ |||||||||
| explicit low 23 bits
|
implied leading 1
not in binary layout
```

Normalized Floating Point: General Case

- A "normalized" floating point number is one like `-248.75 = -1.111100011 * 2^7`
- Leading bit is 1 to indicate negative
- The exponent is in the high order bits in **Bias Form**.
  - Positive integer minus constant **bias number**
  - **Consequence**: exponent of 0 is not bitstring of 0's
  - **Consequence**: tiny exponents like -125 close to bistring of 0's; this makes resulting number close to 0
  - 8-bit exponent `1000 0110 = 128+4+2 = 134` is `133 - 127 = 7`
- The leading 1 before the binary point is **implied** so does not show up in the bit string
- Remaining fractional/mantissa portion shows up in the low-order bits

Bit Ranges/Properties for Parts of IEEE 754 Floats

Kind	Sign	Exponent	Bias	Exp Range	Mantissa
`float`	31 (1)	30-23 (8 bits)	-127	-126 to +127	22-0 (23 bits)
`double`	63 (1)	62-52 (11 bits)	-1023	-1022 to +1023	51-0 (52 bits)

Exercise: Quick Checks

1. Represent 7.125 in binary using "binary point" notation
2. What distinct parts are represented by bits in a floating point number (according to IEEE)
3. What is the "bias" of the exponent for 32-bit `floats`
4. What does the number 1.0 look like as a `float`?

Answers: Quick Checks

1. Represent 7.125 in binary using a "binary point"
   - $7_{10} = 111_2$
   - $0.125_{10} = \frac{1}{8} = 2^{-3} = 0.001_2$
   - $7.125_{10} = 111.001_2$
2. What distinct parts are represented by bits in a floating point number (according to IEEE 754)
   - Sign, Exponent, and Mantissa/Fractional Portion
3. What is the "bias" of the exponent for 32-bit `floats` (according to IEEE 754)
   - Bias is -127 which is subtracted from the unsigned value of the 8 exponent bits to get the actual exponent
4. What does the number 1.0 look like as a `float`?
   - Positive: sign bit of `0`
   - Exponent is 0, so sign bits total 127:
     ```
     0111 1111
     8    4
     ```
   - Mantissa has implied leading 1 and all 0's so:
     ```
     000 0000 0000 0000 0000 0000
     23  20   16   12   8    4
     ```

Special Cases: See `float_bits.c`

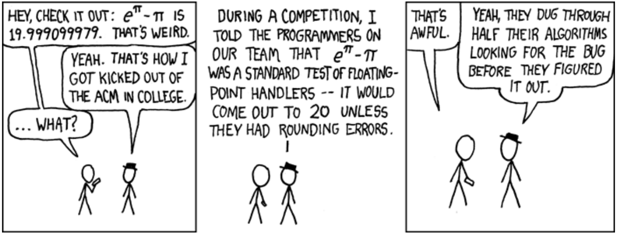
Denormalized values: Exponent bits all 0

- Fractional/Mantissa portion evaluates *without* implied leading one, still an unsigned integer though
- Exponent is $Bias + 1$: 2^{-126} for `float`
- Result: very small numbers close to zero, smaller than any other representation, degrade uniformly to 0
- Zero: bit string of all 0s, optional leading 1 (*negative zero*);

Special Values

- **Infinity**: exponent bits all `1`, fraction all `0`, sign bit indicates $+\infty$ or $-\infty$
- Infinity results from overflow/underflow or certain ops like `float x = 1.0 / 0.0;`
- `#include <math.h>` gets macro `INFINITY` and `-INFINITY`
- **NaN**: not a number, exponent bits all `1`, fraction has some 1s

Other Float Notes



Source: XKCD #217

Approximations and Roundings

- Approximate $\frac{2}{3}$ with 4 digits, usually `0.6667` with standard rounding in base 10
- Similarly, some numbers cannot be exactly represented with fixed number of bits: $\frac{1}{10}$ approximated
- IEEE 754 specifies various rounding modes to approximate numbers

Clever Engineering

- IEEE 754 allows floating point numbers to sort using signed integer routines
- Bit patterns for `float` follows are ordered the same as bit patterns for signed `int`
- Integer comparisons are usually fewer clock cycles than floating comparisons

Sidebar: The Weird and Wonderful Union

- Bitwise operations like & are not valid for `float/double`
- Can use pointers/casting to get around this OR...
- Use a **union**: somewhat unique to C
- Defined like a `struct` with several fields
- BUT fields occupy the same memory location (!?!)
- Allows one to treat a byte position as multiple different types, ex: `int / float / char[]`
- Memory size of the union is the **max** of its fields

```
// union.c
typedef union { // shared memory
  float fl;     // an int
  int in;       // a float
  char ch[4];   // char array
} flint_t;      // 4 bytes total

int main(){
  flint_t flint;
  flint.in = 0xC378C000;
  printf("%.4f\n",    flint.fl);
  printf("%08x %d\n", flint.in);
  for(int i=0; i<4; i++){
    unsigned char c = flint.ch[i];
    printf("%d: %02x '%c'\n",i,c,c);
  }
}
```

Symbol	Mem	Val
flint.ch[3]	#1027	0xC3
flint.ch[2]	#1026	0x78
flint.ch[1]	#1025	0xC0
flint.in/fl/ch[0]	#1024	0x00
i	#1020	?

Floating Point Operation Efficiencies

- Floating Point Operations per Second, **FLOPS** is a major measure for numerical code/hardware efficiency
- Often used to benchmark and evaluate scientific computer resources, (e.g. top super computers in the world)
- Tricky to evaluate because of
  - A single FLOP (add/sub/mul/div) may take 3 clock cycles to finish: **latency 3**
  - Another FLOP **can start** before the first one finishes: **pipelined**
  - Enough FLOPs lined up can get **average 1 FLOP per cycle**
  - FP Instructions may automatically operate on multiple FPs stored in memory to feed pipeline: **vectorized ops**
  - Generally referred to as **superscalar**
  - Processors schedule things **out of order** too
- All of this makes micro-evaluation error-prone and pointless
- Run a real application like an N-body simulation and compute

$$\text{FLOPS} = \frac{\text{number of floating ops done}}{\text{time taken in seconds}}$$

Top 5 Super Computers Worldwide, Nov 2017

Rank	System	#Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	Sunway TaihuLight *China* Sunway MPP	10,649,600	93,014.6	125,435.9	15,371
2	Tianhe-2 (MilkyWay-2) *China* TH-IVB-FEP Cluster	3,120,000	33,862.7	54,902.4	17,808
3	Piz Daint *Switzerland* Cray XC50	361,760	19,590.0	25,326.3	2,272
4	Gyoukou *Japan* ZettaScaler-2.2 HPC system	19,860,000	19,135.8	28,192.0	1,350
5	Titan *USA* Cray XK7	560,640	17,590.0	27,112.5	8,209

https://www.top500.org/lists/2017/11/