CMSC216: Binary Floating Point Numbers

Chris Kauffman

Last Updated:
Fri Feb 28 02:49:46 PM EST 2025

Logistics

Reading Bryant/O'Hallaron

- Ch 2.1-3: Integers
- Ch 2.4-5: Floats (Optional)
- Quick Guide to GDB

Goals

- Finish Ints / Bitwise Ops
- Brief: Floating Point layout
- Thu: Assembly

Assignments

- Lab05: Bits and GDB
- HW05: Assembly Intro
- Project 2: Bitwise Ops, GDB, C Application

P2 will go up within the next day

Grading on Exam 1 / Project 1 ongoing, release grades towards end of week

Announcements

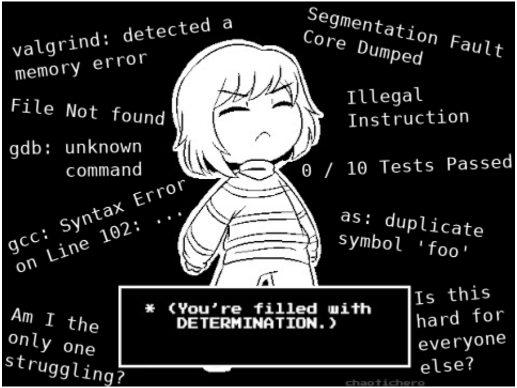
Midterm Feedback Survey

- Available on Canvas; Anonymous Feedback
- Worth 1 Full Engagement Point (like labs)
- Due 11:59pm Fri 07-Mar-2025

Exam 1 Makeup

- Prof K has emailed all students with permission to make up exam 1 about scheduling
- If you expected to take the makeup exam and have not heard from Prof K **email him ASAP**

Don't Give Up, Stay Determined!



- If Project 1 / Exam 1 went awesome, count yourself lucky
- If things did not go well, Don't Give Up
- Spend some time contemplating why things didn't go well, talk to course staff about it, learn from mistakes
- There is a LOT of semester left and plenty of time to recover from a bad start

Note on Float Coverage

- Floating point layout is complex and interesting but...
- It's not a core topic that will appear on any exams, only tangentially on assignments
- Our coverage will be brief, examine slides / textbook if you want more depth
- **GOAL:** Demonstrate that (1) Real numbers can be approximated and (2) doing so uses bits in a very different way than integer representations

Parts of a Fractional Number

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

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

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

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

One *could* represent fractional numbers with a **fixed point** e.g.

- 32 bit fractional number with
- 10 bits left of Binary Point (integer part)
- 22 bits right of Binary Point (fractional part)

BUT most applications require a more flexible scheme

Scientific Notation for Numbers

"Scientific" or "Engineering" notation for numbers with a fractional part is

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 **non-zero** digit left of 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 $\equiv$ Significand $\equiv$ Fractional Part

Principle and Practice of Binary Floating Point Numbers

- In early computing, computer manufacturers used similar principles for floating point numbers but varied specifics
- Example of Early float data/hardware
  - Univac: 36 bits, 1-bit sign, 8-bit exponent, 27-bit significand[1]
  - IBM: 32 bits, 1-bit sign, 7-bit exponent, 24-bit significand[2]
- Manufacturers implemented circuits with different rounding behavior, with/without infinity, and other inconsistencies
- Troublesome for reliability: code produced different results on different machines
- This was resolved with the adoption of the IEEE 754 Floating Point Standard which specifies
  - Bit layout of 32-bit `float` and 64-bit `double`
  - Rounding behavior, special values like Infinity
- Turing Award to William Kahan for his work on the standard

---

[1] Floating Point Arithmetic

[2] IBM Hexadecimal Floats

IEEE 754 Format: *The* Standard for Floating Point

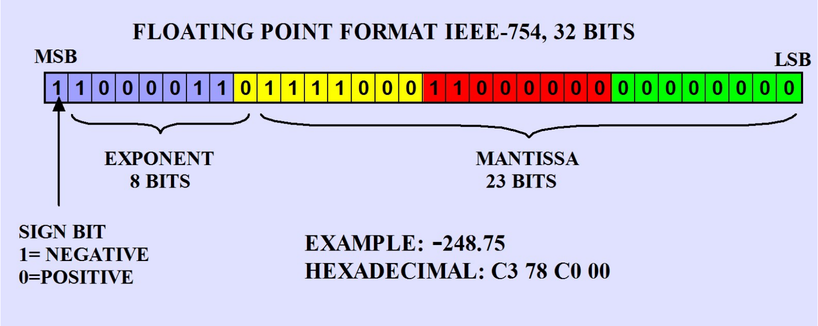
`float`	`double`	Property
32	64	Total bits
1	1	Bits for sign (1 neg / 0 pos)
8	11	Bits for Exponent multiplier (power of 2)
23	52	Bits for Fractional part or **mantissa**
7.22	15.95	Decimal digits of accuracy[3]

- Most commonly implemented format for floating point numbers in hardware to do arithmetic: processor has physical circuits to add/mult/etc. for this bit layout of floats
- Numbers/Bit Patterns divided into three categories

Category	**Description**	**Exponent**
Normalized	most common like `1.0` and `-9.56e37`	mixed 0/1
Denormalized	very close to zero and `0.0`	all 0's
Special	extreme/error values like `Inf` and `NaN`	all 1's

[3] Wikipedia: IEEE 754

Example `float` Layout of -248.75: `float_examples.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 in the standard range for `float/double`, bit layout follows previous slide
- Example: `-248.75 = -1.111100011 * 2^7`

Exponent is in **Bias Form** (not Two's Complement)

- Unsigned positive integer minus constant **bias number**
- **Consequence**: exponent of 0 is not bitstring of 0's
- **Consequence**: tiny exponents like -125 close to bitstring of 0's; this makes resulting number close to 0
- 8-bit exponent `1000 0110 = 128+4+2 = 134`
  so exponent value is `134 - 127 = 7`

Integer and Mantissa Parts

- 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

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 construct 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

```c
// union.c
typedef union { // shared memory
  float fl;     // float 4 bytes
  int in;       // int   4 bytes
  char ch[4];   // array 4 bytes
} flint_t;      // 4 bytes total (?!)
// all fields are in the same memory
// so max of (4,4,4) rather than sum

int main(){
  flint_t flint;
  flint.in = 0xC378C000;
  printf("%.4f\n",   flint.fl);
  printf("%08x %d\n",flint.in,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 | ?    |
```

======== OPTIONAL MATERIAL ========

The remaining material will be discussed time permitting but is oriented towards those with deeper curiosity and will not feature in assignments / exams

Fixed Bit Standards for Floating Point

IEEE Standard Layouts

Kind	Sign Bit	Exponent Bits	Bias	Exp Range	Mantissa Bits
`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)

Standard allows hardware to be created that is as efficient as possible to do calculation on these numbers

Consequences of Fixed Bits

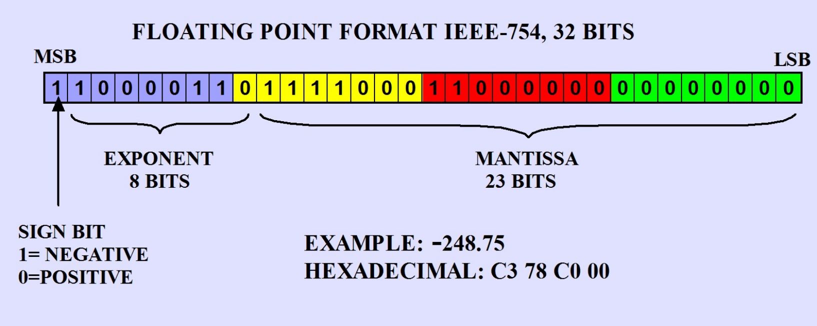
- Since a fixed # of bit is used, **some numbers cannot be exactly represented**, happens in any numbering system:
- Base 10 and Base 2 cannot represent $\frac{1}{3}$ in finite digits
- Base 2 cannot represent $\frac{1}{10}$ in finite digits

  ```
  float f = 0.1;
  printf("0.1 = %.20e\n",f);
  0.1 = 1.00000001490116119385e-01
  ```

  Try `show_float.c` to see this in action

Exercise: Quick Checks

1. What distinct parts are represented by bits in a floating point number (according to IEEE)
2. What is the "bias" of the exponent for 32-bit `floats`
3. Represent 7.125 in binary using "binary point" notation
4. Lay out 7.125 in IEEE-754 format
5. What does the number 1.0 look like as a `float`?



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

The diagram above may help in recalling IEEE 754 layout