The Hardware/Software Interface

CSE351 Winter 2011

Module 4: Floating Point
(but nearly nothing about C pointers)

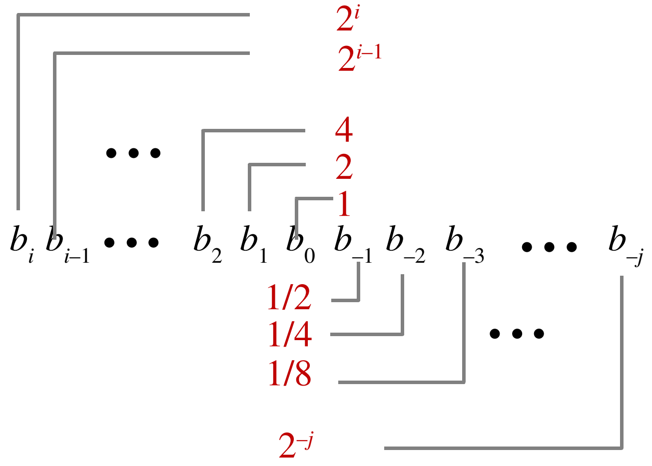
Today Topics: Floating Point

- **Background: Fractional binary numbers**
- **IEEE floating point standard: Definition**
- **Example and properties**
- **Rounding, addition, multiplication**
- **Floating point in C**
- **Summary**

(Abstract) Fractional binary numbers

- **What is 1011.101?**

Fractional Binary Numbers



2^i

2^{i-1}

• • •

4

2

1

$b_i \; b_{i-1} \; \bullet\bullet\bullet \; b_2 \; b_1 \; b_0 \; b_{-1} \; b_{-2} \; b_{-3} \; \bullet\bullet\bullet \; b_{-j}$

1/2

1/4

1/8

• • •

2^{-j}

- **Representation**
  - Bits to right of “binary point” represent fractional powers of 2
  - Represents rational number:

$$\sum_{k=-j}^{i} b_k \cdot 2^k$$

Fractional Binary Numbers: Examples

Value	**Representation**
5 and 3/4	`101.11`
2 and 7/8	`10.111`
0 and 23/32	`0.10111`

Issue #1: Representable Numbers

- **Limitation**
  - Even given an arbitrary number of bits, can only exactly represent numbers of the form *x*/2*k*
  - Other rational numbers have repeating bit representations

Value	**Representation**
1/3	`0.0101010101[01]`
1/5	`0.001100110011[0011]`
1/10	`0.0001100110011[0011]`

Fixed Point Representation

- float → 32 bits; double → 64 bits
- We might try representing fractional binary numbers by picking a fixed place for an implied binary point
  - "fixed point binary numbers"
- Let's do that, using 8 bit floating point numbers as an example
  - #1: the binary point is between bits 2 and 3
    $b_7\ b_6\ b_5 b_4\ b_3$ [.] $b_2\ b_1\ b_0$
  - #2: the binary point is between bits 4 and 5
    $b_7\ b_6\ b_5$ [.] $b_4\ b_3\ b_2\ b_1\ b_0$
  - The position of the binary point affects the <u>range</u> and <u>precision</u>
    - range: difference between the largest and smallest representable numbers
    - precision: smallest possible difference between any two numbers

Fixed Point Pros and Cons

- Pros
  - It's simple. The same hardware that does integer arithmetic can do fixed point arithmetic
    - In fact, the programmer can use ints with an implicit fixed point
      - E.g., int balance; // number of pennies in the account
    - ints are just fixed point numbers with the binary point to the right of b_0
- Cons
  - There is no good way to pick where the fixed point should be
    - Sometimes you need range, sometimes you need precision. The more you have of one, the less of the other
- Fixing fixed point representation: floating point
- Do that in a way analogous to "'scientific notation"
  - Not 12000000, but 1.2×10^{7}
    Not 0.0000012, but 1.2×10^{-6}

Floating Point

- Abstractly, floating point is analogous to scientific notation
  - Decimal:
    - Not 12000000, but 1.2 x 10^7
      Not 0.0000012, but 1.2 x 10^{-6}
  - Binary:
    - Not 11000.000, but 1.1 x 2^4
      Not 0.000101, but 1.01 x 2^{-4}

- We have to divvy up the bits we have (e.g., 32) among:
  - the sign (1 bit)
  - the significand
  - the exponent

IEEE Floating Point

- **IEEE Standard 754**
  - Established in 1985 as uniform standard for floating point arithmetic
  - Main idea: make numerically sensitive programs portable
  - Specifies two things: representation and result of floating operations
  - IEEE 754 now supported by all major CPUs

- **Driven by numerical concerns**
  - Numerical analysts predominated over hardware designers in defining standard
  - Nice standards for rounding, overflow, underflow, but...
    - But... hard to make fast in hardware
  - Float operations can be an order of magnitude slower than integer

Floating Point Representation

- **Numerical Form:**

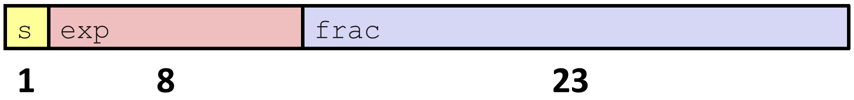
  $(-1)^s\ M\ 2^E$

  - **Sign bit** ***s*** determines whether number is negative or positive
  - **Significand (mantissa)** ***M*** normally a fractional value in range [1.0,2.0).
  - **Exponent** ***E*** weights value by power of two

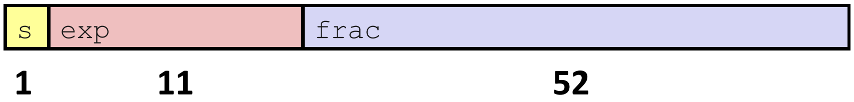
- **Encoding**
  - MSB **`s`** is sign bit ***s***
  - **`frac`** field encodes ***M*** (but is not equal to M)
  - **`exp`** field encodes ***E*** (but is not equal to E)

Precisions

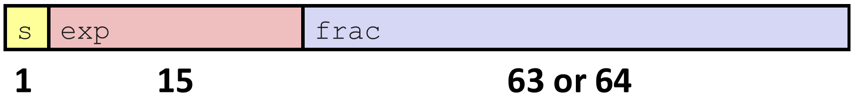
- **Single precision: 32 bits** (largest value: about 3.4×10^{38})



- **Double precision: 64 bits** (largest value: about 1.8×10^{308})



- **Extended precision: 80 bits (Intel only)** (largest: about 1.2×10^{4932})



Normalization and Special Values

- “Normalized” means mantissa has form 1.xxxxx
  - 0.011 x 2^5 and 1.1 x 2^3 represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, don't bother to store it

- Special values:
  - The float value 00...0 represents zero
  - If the exp == 11...1 and the mantissa == 00...0, it represents ∞
    - E.g., 10.0 / 0.0 → ∞
  - If the exp == 11...1 and the mantissa != 00...0, it represents NaN
    - “Not a Number”
    - Results from operations with undefined result
      - E.g., 0 * ∞

Floating Point and the Programmer

```
#include <stdio.h>

int main(int argc, char* argv[]) {

  float f1 = 1.0;
  float f2 = 0.0;
  int i;
  for ( i=0; i<10; i++ ) {
    f2 += 1.0/10.0;
  }

  printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
  printf("f1 = %10.8f\n", f1);
  printf("f2 = %10.8f\n\n", f2);

  f1 = 1E30;
  f2 = 1E-30;
  float f3 = f1 + f2;
  printf ("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );

  return 0;
}
```

```
$ ./a.out
0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119

f1 == f3? yes
```

Summary

- As with integers, floats suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow, just like ints
  - Some “simple fractions” have no exact representation
    - E.g., 0.1
  - Can also lose precision, unlike ints
    - “Every operation gets a slightly wrong result”
- Mathematically equivalent ways of writing an expression may compute differing results
- NEVER test floating point values for equality!