Today: Floats!

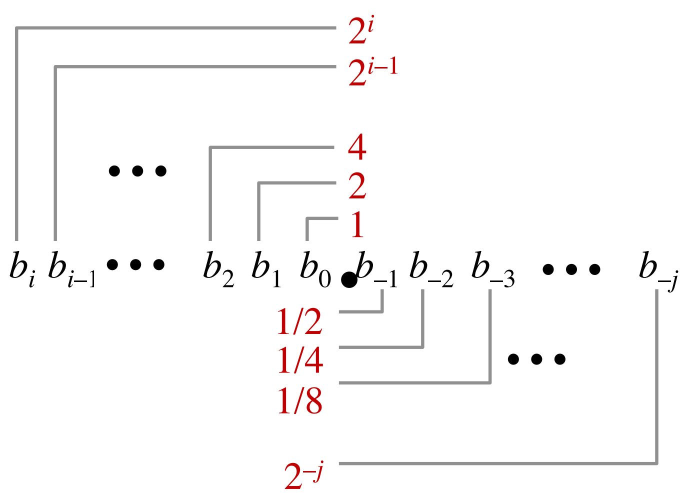
Today Topics: Floating Point

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

Fractional binary numbers

- **What is 1011.101?**

Fractional Binary Numbers



- **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**

Value	Representation
5 and 3/4	101.11_2
2 and 7/8	10.111_2
63/64	0.111111_2

- **Observations**
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Representable Numbers

- **Limitation**
  - Can only exactly represent numbers of the form $x/2^k$
  - Other rational numbers have repeating bit representations

- **Value** **Representation**

Value	Representation
1/3	$0.0101010101[01]\ldots_2$
1/5	$0.001100110011[0011]\ldots_2$
1/10	$0.0001100110011[0011]\ldots_2$

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

What else could we do?

IEEE Floating Point

- **Fixing fixed point: analogous to scientific notation**
  - Not 12000000 but 1.2 x 10^7; not 0.0000012 but 1.2 x 10^-6
- **IEEE Standard 754**
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs
- **Driven by numerical concerns**
  - Nice standards for rounding, overflow, underflow
  - Hard to make fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard

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)**

s	exp	frac

Precisions

- **Single precision: 32 bits**

s	exp	frac
1	8	23

- **Double precision: 64 bits**

s	exp	frac
1	11	52

- **Extended precision: 80 bits (Intel only)**

s	exp	frac
1	15	63 or 64

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
- **How do we do 0? How about 1.0/0.0?**

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 * ∞

How do we do operations?

- **Is representation exact?**
- **How are the operations carried out?**

Floating Point Operations: Basic Idea

- $x +_f y$ = Round(x + y)

- $x *_f y$ = Round(x * y)

- **Basic idea**
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into `frac`

Floating Point Multiplication

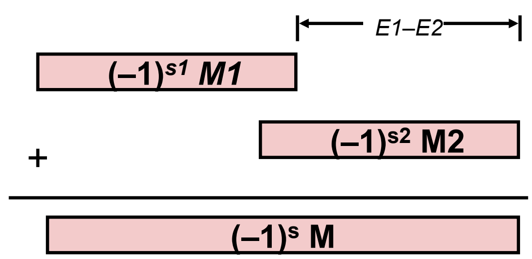
$(-1)^{s1}\ M1\ \ 2^{E1}\ \ *\ \ (-1)^{s2}\ M2\ \ 2^{E2}$

- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign *s*: *s1 ^ s2*
  - Significand *M*: *M1 * M2*
  - Exponent *E*: *E1 + E2*
- **Fixing**
  - If $M \geq 2$, shift *M* right, increment *E*
  - If *E* out of range, overflow
  - Round *M* to fit `frac` precision
- **Implementation**
  - What is hardest?

Floating Point Addition

$(-1)^{s1}\ M1\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\ M2\ \ 2^{E2}$

Assume *E1 > E2*



- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign *s*, significand *M*:
    - Result of signed align & add
  - Exponent *E*: *E1*
- **Fixing**
  - If $M \geq 2$, shift *M* right, increment *E*
  - if $M < 1$, shift *M* left *k* positions, decrement *E* by *k*
  - Overflow if *E* out of range
  - Round *M* to fit `frac` precision

Hmm… if we round at every operation…

Mathematical Properties of FP Operations

- **Not really** associative or distributive due to rounding
- **Infinities and NaNs cause issues**
- **Overflow and infinity**

Floating Point in C

- **C Guarantees Two Levels**

  `float` single precision

  `double` double precision

- **Conversions/Casting**
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representation
  - **`Double`/`float`** → **`int`**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - **`int`** → **`double`**
    - Exact conversion, why?
  - **`int`** → **`float`**
    - Will round according to rounding mode

University of Washington

Memory Referencing Bug (Revisited)

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

```
fun(0)   ->     3.14
fun(1)   ->     3.14
fun(2)   ->     3.1399998664856
fun(3)   ->     2.00000061035156
fun(4)   ->     3.14, then segmentation fault
```

Explanation:

`Saved State`	4
`d7 … d4`	3
`d3 … d0`	2
`a[1]`	1
`a[0]`	0

Location accessed by `fun(i)`

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;
}
```


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!