Representing and Manipulating Floating Points

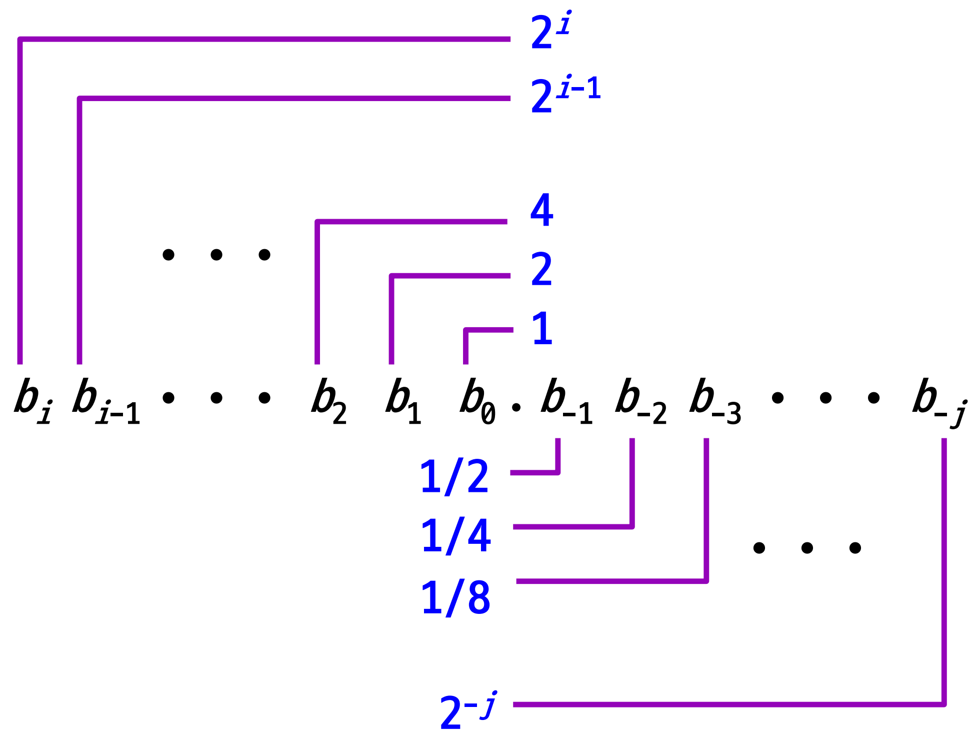
Jo, Heeseung

The Problem

How to represent fractional values with finite number of bits?

- 0.1
- 0.612
- 3.14159265358979323846264338327950288...

Fractional Binary Numbers (1)



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 (2)

Examples:

Value	Representation
5 + 3/4	101.11_2
2 + 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 $0.111111\ldots_2$ just below 1.0
  - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
  - Use notation $1.0 - \varepsilon$

Fractional Binary Numbers (3)

Representable numbers

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

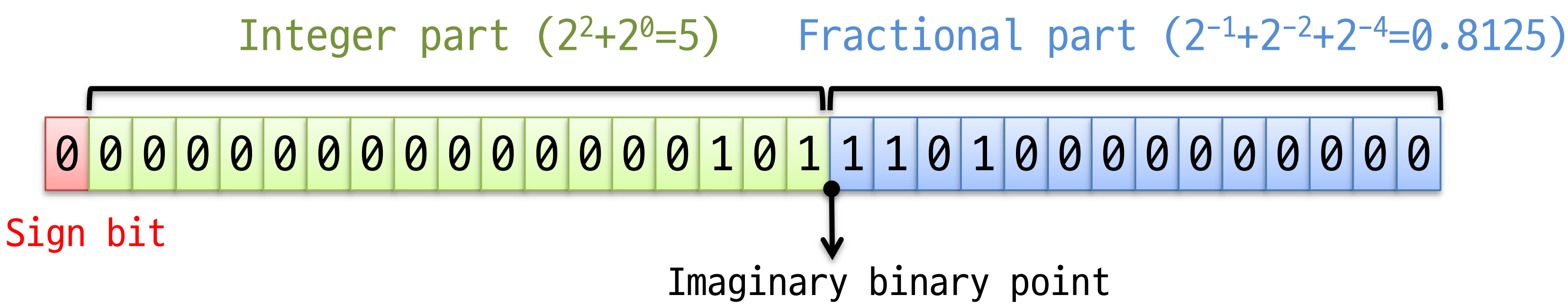
Value	Representation
1/3	$0.0101010101[01]\ldots_2$
1/5	$0.001100110011[0011]\ldots_2$
1/10	$0.0001100110011[0011]\ldots_2$

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

Fixed-Point Representation (1)

p.q Fixed-point representation

- Use the rightmost ***q*** bits of an integer as representing a fraction
- Example: 17.14 fixed-point representation
  - 1 bit for sign bit
  - 17 bits for the integer part
  - 14 bits for the fractional part
  - An integer ***x*** represents the real number ***x*** / 2^{14}
  - Maximum value: $(2^{31} - 1) / 2^{14} \fallingdotseq 131071.999$



Fixed-Point Representation (2)

> *x*, *y*: fixed-point number
> *n*: integer
> *f* = 1 << *q*

Properties

- Convert *n* to fixed point: $n * f$
- Add *x* and *y* : $x + y$

```
  0 0000000000000 1101 . 10000000000000   13.5
+ 0 0000000000000 0001 . 11000000000000    1.75
-----------------------------------------------
  0 0000000000000 1111 . 01000000000000   15.25
```

- Subtract *y* from *x* : $x - y$
- Add *x* and *n* : $x + n * f$
- Multiply *x* by *n* : $x * n$
- Divide *x* by *n* : x / n

Fixed-Point Representation (3)

Pros

- Simple
- Can use integer arithmetic to manipulate
- No floating-point hardware needed
- Used in many low-cost embedded processors or DSPs (digital signal processors)

Cons

- Cannot represent wide ranges of numbers
  - 1 Light-Year = 9,460,730,472,580.8 km
  - The radius of a hydrogen atom: 0.000000000025 m

Is it the best?

Representing Floating Points

IEEE standard 754

- Established in 1985 as uniform standard for floating point arithmetic
  - Before that, many idiosyncratic formats

- Supported by all major CPUs
- William Kahan, a primary architect of IEEE 754, won the Turing Award in 1989
- Driven by numerical concerns
  - Nice standards for rounding, overflow, underflow
  - Hard to make go fast

Normalized Form

Like scientific notation

- -2.34×10^{56} ← normalized
- $+0.002 \times 10^{-4}$ ← not normalized
- $+987.02 \times 10^{9}$ ← not normalized

In binary

- $\pm 1.xxxxxxx_2 \times 2^{yyyy}$

FP Representation

Numerical form: -1^{s} x 1.M x 2^{E}

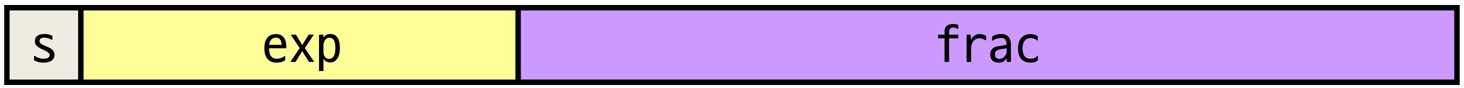
- Sign bit s determines whether number is negative or positive
- Significand M normally a fractional value in range [1.0,2.0)
- Exponent E weights value by power of two

Encoding

- **MSB** is sign bit
- **exp** field encodes E (Exponent)
- **frac** field encodes M (Mantissa)

FP Precisions

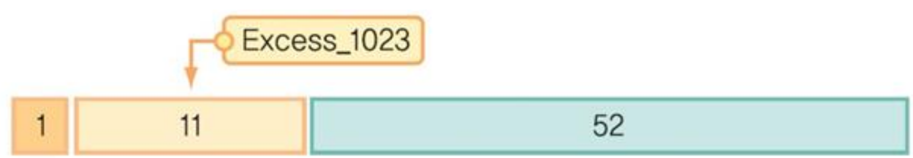
Encoding



- **MSB** is sign bit
- **exp** field encodes E (Exponent)
- **frac** field encodes M (Mantissa)

Sizes

- Single precision: 8 exp bits, 23 frac bits (32 bits total)
- Double precision: 11 exp bits, 52 frac bits (64 bits total)
- Extended precision: 15 exp bits, 63 frac bits
  - Only found in Intel-compatible machines
  - Stored in 80 bits (1 bit wasted)



Excess notation

Ex. 8 excess notation for 4 bit

- Each value in excess notation of its original value in binary

Exponent part uses excess notation

- Simpler than 2's complement

Bit pattern	Value represented
1111	7
1110	6
1101	5
1100	4
1011	3
1010	2
1001	1
1000	0
0111	−1
0110	−2
0101	−3
0100	−4
0011	−5
0010	−6
0001	−7
0000	−8

Normalized Values (1)

Condition: exp ≠ 000…0 and exp ≠ 111…1

Exponent coded as biased value (127/1023 excess notation)

- ***E* = *Exp* – *Bias***
- ***Exp*** : unsigned value denoted by **exp**
- ***Bias*** : Bias to represent a negative value
  - Single precision: 127 (***Exp*** : 1..254, ***E*** : -126..127)
  - Double precision: 1023 (***Exp*** : 1..2046, ***E*** : -1022..1023)

Significand coded with implied leading 1

- $M = 1.xxx\ldots x_2$
  - Minimum when 000…0 (*M* = 1.0)
  - Maximum when 111…1 (*M* = 2.0 – ε)
- Get extra leading bit for "free"

Normalized Values (2)

Value: **float f = 2003.0;**

2003_{10} = 11111010011_2 = $1.1111010011_2 \times 2^{10}$

Significand

M = 1.1111010011_2

Frac = $11110100110000000000000_2$

Exponent

E = 10

Exp = *E + Bias* = 10 + 127 = 137 = 10001001_2

Floating Point Representation:

2003:			111	1010	011			
137:	100	0100	1					
Binary:	0100	0100	1111	1010	0110	0000	0000	0000
Hex:	4	4	F	A	6	0	0	0

Floating-Point Example

Represent -0.75

- $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
- S = 1
- Fraction = $1000\ldots00_2$
- Exponent = -1 + Bias
  - Single: -1 + 127 = 126 = 01111110_2
  - Double: -1 + 1023 = 1022 = 01111111110_2

Single: 1011 1111 0100 0…00

Double: 1011 1111 1110 1000 0…00

Floating-Point Example

What number is represented by the single-precision float

11000000101000…00

- S = 1
- Fraction = $01000…00_2$
- Exponent = 10000001_2 = 129

$x = (-1)^1 \times (1 + 01_2) \times 2^{(129\ -\ 127)}$

$= (-1) \times 1.25 \times 2^2$

$= -5.0$

Denormalized Values

Condition: exp = 000…0

Cases

- **exp = 000…0, frac = 000…0**
  - Represents value 0
  - Note that have distinct values +0 and –0
- **exp = 000…0, frac ≠ 000…0**
  - Numbers very close to 0.0

Special Values

Condition: exp = 111…1

Cases

- **exp = 111…1, frac = 000…0**
  - Represents value + or −∞(infinity)
  - Operation that overflows
  - Both positive and negative
  - e.g. 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞
- **exp = 111…1, frac ≠ 000…0**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - e.g., sqrt(−1), ∞ − ∞, ∞ x 0, ...

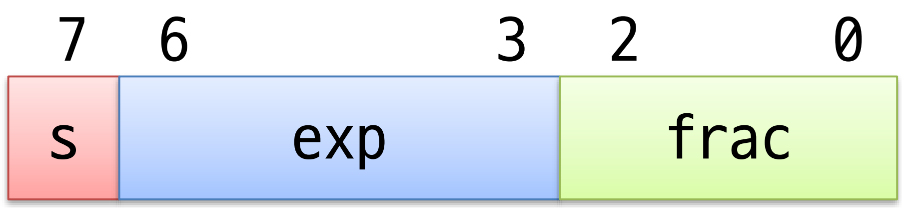
Tiny FP Example (1)

8-bit floating point representation

- The sign bit is in the most significant bit
- The next four bits are the **exp**, with a bias of 7
- The last three bits are the **frac**

Same general form as IEEE format

- Normalized, denormalized
- Representation of 0, NaN, infinity



Interesting Numbers

Description	exp	frac	Numeric Value
Zero	000 … 00	000 … 00	0.0
Smallest Positive Denormalized	000 … 00	000 … 01	Single: $2^{-23} \times 2^{-126} \approx 1.4 \times 10^{-45}$ Double: $2^{-52} \times 2^{-1022} \approx 4.9 \times 10^{-324}$
Largest Denormalized	000 … 00	111 … 11	Single: $(1.0 - \varepsilon) \times 2^{-126} \approx 1.18 \times 10^{-38}$ Double: $(1.0 - \varepsilon) \times 2^{-1022} \approx 2.2 \times 10^{-308}$
Smallest Positive Normalized	000 … 01	000 … 00	Single: 1.0×2^{-126}, Double: 1.0×2^{-1022} (Just larger than largest denormalized)
One	011 … 11	000 … 00	1.0
Largest Normalized	111 … 10	111 … 11	Single: $(2.0 - \varepsilon) \times 2^{127} \approx 3.4 \times 10^{38}$ Double: $(2.0 - \varepsilon) \times 2^{1023} \approx 1.8 \times 10^{308}$

Special Properties

FP zero same as integer zero

- All bits = 0

Can (almost) use unsigned integer comparison

- Must first compare sign bits
- Must consider -0 = 0
- NaNs problematic
  - Will be greater than any other values
- Otherwise OK
  - Denormalized vs. normalized
  - Normalized vs. Infinity

0001 1100 0011 1111 0000 1010 1101 0101

vs.

0001 0101 1101 1111 1111 1111 1111 1110

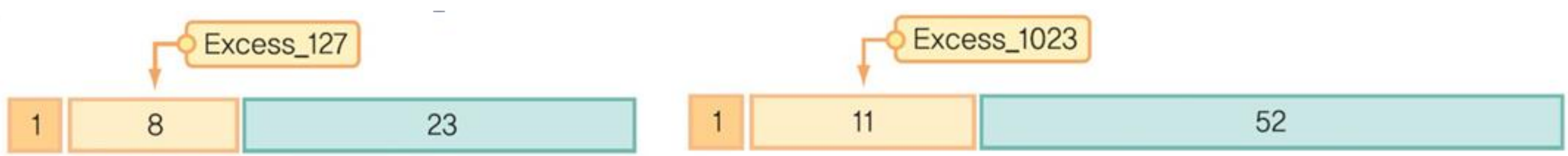
Floating Point in C (1)

C guarantees two levels

- **float** (single precision) vs. **double** (double precision)

Conversions

- **double** or **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, as long as **int** has ≤ 53 bit word size
- **int** -> **float**
  - Will round according to rounding mode



Floating Point in C (2)

Example 1:

```
#include <stdio.h>

int main ()  {
      int n = 123456789;
      int nf, ng;
      float f;
      double g;

      f = (float) n;
      g = (double) n;
      nf = (int) f;
      ng = (int) g;
      printf ("nf=%d ng=%d\n", nf, ng);
}
```

Floating Point in C (3)

Example 2:

```
#include <stdio.h>

int main ()  {
      double  d;

      d = 1.0 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
              + 0.1 + 0.1 + 0.1 + 0.1 + 0.1;

      if (d==2.0)
          printf("true 1\n");
      if (d==2)
          printf("true 2\n");
      if ((int)d==2)
          printf("true 3\n");
      printf ("d = %.20f\n", d);
}
```

Floating Point in C (4)

Example 3:

```
#include <stdio.h>

int main ()  {
      float f1 = (3.14 + 1e20) - 1e20;
      float f2 = 3.14 + (1e20 - 1e20);

      printf ("f1 = %f, f2 = %f\n", f1, f2);
}
```

Ariane 5

Ariane 5 tragedy (June 4, 1996)

- Exploded 37 seconds after liftoff
- Satellites worth $500 million

Why?

- Computed horizontal velocity as floating point number
- Converted to 16-bit integer
  - Careful analysis of Ariane 4 trajectory proved 16-bit is enough
- Reused a module from 10-year-old s/w
  - Overflowed for Ariane 5
  - No precise specification for the S/W
- Refer more
  - http://www.youtube.com/watch?v=gp_D8r-2hwk
  - http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html

Summary

IEEE floating point has clear mathematical properties

- Represents numbers of form $M \times 2^{E}$
- Can reason about operations independent of implementation
  - As if computed with perfect precision and then rounded
- Not the same as real arithmetic
  - Violates associativity/distributivity
  - Makes life difficult for compilers and serious numerical applications programmers