Representing and Manipulating Floating Points

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
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:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-3/4</td>
<td>101.11₂</td>
</tr>
<tr>
<td>2-7/8</td>
<td>10.111₂</td>
</tr>
<tr>
<td>63/64</td>
<td>0.111111₁₀₂</td>
</tr>
</tbody>
</table>

- **Observations**
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of form \(0.\underbrace{111111\ldots}_2\) just below 1.0
    - \(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^i} + \ldots \rightarrow 1.0\)
    - Use notation \(1.0 - \varepsilon\)
### Fractional Binary Numbers (3)

- **Representable numbers**
  - Can only exactly represent numbers of the form $\frac{x}{2^k}$
  - Other numbers have repeating bit representations

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>0.010101010101[01]...$_2$</td>
</tr>
<tr>
<td>1/5</td>
<td>0.001100110011[0011]...$_2$</td>
</tr>
<tr>
<td>1/10</td>
<td>0.0001100110011[0011]...$_2$</td>
</tr>
</tbody>
</table>
**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: $(2^{31} - 1) / 2^{14} \approx 131071.999$
### Fixed-Point Representation (2)

#### Properties

- Convert $n$ to fixed point: $n \times f$
- Add $x$ and $y$: $x + y$
- Subtract $y$ from $x$: $x - y$
- Add $x$ and $n$: $x + n \times f$
- Multiply $x$ by $n$: $x \times n$
- Divide $x$ by $n$: $x / n$

$x, y$: fixed-point number
$n$: integer
$f = 1 \ll q$
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
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
    - Numerical analysts predominated over hardware types in defining standard.
FP Representation

- **Numerical form:** \(-1^s \times M \times 2^E\)
  - 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 (32bits total)
  - Double precision: 11 `exp` bits, 52 `frac` bits (64bits total)
  - Extended precision: 15 `exp` bits, 63 `frac` bits
    - Only found in Intel-compatible machines
    - Stored in 80 bits (1 bit wasted)
Normalized Values (1)

- **Condition:** $\text{exp} \neq 000...0$ and $\text{exp} \neq 111...1$
- **Exponent coded as biased value**
  - $E = \text{Exp} - \text{Bias}$
  - $\text{Exp}$: unsigned value denoted by $\text{exp}$
  - $\text{Bias}$: Bias value
    - Single precision: 127 ($\text{Exp} : 1..254, \text{E} : -126..127$)
    - Double precision: 1023 ($\text{Exp} : 1..2046, \text{E} : -1022..1023$)
- **Significand coded with implied leading 1**
  - $M = 1.xxx...x_2$
    - Minimum when 000...0 ($M = 1.0$)
    - Maximum when 111...1 ($M = 2.0 - \varepsilon$)
  - Get extra leading bit for “free”
Normalized Values (2)

- **Value:** \( \text{float } f = 2003.0; \)
  
  \[ 2003_{10} = 11111010011_2 = 1.1111010011_2 \times 2^{10} \]

- **Significand**
  
  \[ M = 1.1111010011_2 \]
  
  \[ \text{frac} = 1111010011000000000000000000_2 \]

- **Exponent**
  
  \[ E = 10 \]
  
  \[ \text{Exp} = E + \text{Bias} = 10 + 127 = 137 = 10001001_2 \]

**Floating Point Representation:**

- **Hex:** 4 4 F A 6 0 0 0 0 0 0
- **Binary:** 0100 0100 1111 1010 0110 0000 0000 0000
- **137:** 100 0100 1
- **2003:** 1111 1010 0110
Denormalized Values

- **Condition:** \( \text{exp} = 000...0 \)

- **Value**
  - Exponent value \( E = 1 - \text{Bias} \)
  - Significand value \( M = 0.xxx...x_2 \) (no implied leading 1)

- **Cases**
  - \( \text{exp} = 000...0, \frac{\text{frac}}{\text{exp}} = 000...0 \)
    - Represents value 0
    - Note that have distinct values +0 and -0
  - \( \text{exp} = 000...0, \frac{\text{frac}}{\text{exp}} \neq 000...0 \)
    - Numbers very close to 0.0
    - “Gradual underflow”: possible numeric values are spaced evenly near 0.0
Special Values

- **Condition:** exp = 111...1

- **Cases**
  - exp = 111...1, frac = 000...0
    - Represents value $\infty$ (infinity)
    - Operation that overflows
    - Both positive and negative
    - e.g. 1.0/0.0 = -1.0/-0.0 = $+\infty$, 1.0/-0.0 = $-\infty$
  - exp = 111...1, frac $\neq$ 000...0
    - Not-a-Number (NaN)
    - Represents case when no numeric value can be determined
    - e.g., $\sqrt{-1}$, $\infty - \infty$, $\infty \times 0$, ...
## Interesting Numbers

<table>
<thead>
<tr>
<th>Description</th>
<th>exp</th>
<th>frac</th>
<th>Numeric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>000 ... 00</td>
<td>000 ... 00</td>
<td>0.0</td>
</tr>
</tbody>
</table>
| 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: $1.0 \times 2^{-126}$, Double: $1.0 \times 2^{-1022}$ (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
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
Example 1:

```c
#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;

    printf ("d = %.20f\n", d);
}

Floati ng Point in C (4)

- Example 3:

```c
#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
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