Floating Point
Most slides from CMU 213
Representation

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

\[ \sum_{k=-j}^{i} b_k \cdot 2^k \]
Frac. Binary Number 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>
</tbody>
</table>

Observations

- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form 0.11111₁₂ 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 $\frac{x}{2^k}$
- 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>

This is where we need to slightly sacrifice precision when storing these numbers
Floating Point Representation

**Numerical Form**

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

**Example:**

\(15213_{10} = (-1)^0 \times 1.1101101101101_2 \times 2^{13}\)

**Encoding**

- **MSB** is sign bit
- **exp field encodes** \(E\) (but is not equal to \(E\))
- **frac field encodes** \(M\) (but is not equal to \(M\))
IEEE Floating Point

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 go fast
  - Numerical analysts predominated over hardware types in defining standard

Drawback
- Naturally, it cannot represent all real numbers accurately
  - It cannot store recurring digits in base 10 like 1/3, so these numbers are always rounded down slightly. It allows use to store very small and also large numbers by reducing a little precision.
IEEE Precision Options

Numbers are stored in scientific notation

◊ Single precision: 32 bits  —> approximately ± 10 ^ 38

◊ Double precision: 64 bits —> approximately ± 10 ^ 308
“Normalized” Numeric Values

Condition
- \( \text{exp} \neq 000\ldots0 \) and \( \text{exp} \neq 111\ldots1 \)

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\ldots254, \( E \): -126\ldots127)
  - Double precision: 1023 (\( \text{Exp} \): 1\ldots2046, \( E \): -1022\ldots1023)
  - in general: \( \text{Bias} = 2^{e-1} - 1 \), where \( e \) is number of exponent bits

Significand coded with implied leading 1
- \( M = 1.xxx\ldots x_2 \)
  - \( xxx\ldots x \): bits of frac
  - Minimum when 000\ldots0 (\( M = 1.0 \))
  - Maximum when 111\ldots1 (\( M = 2.0 - \varepsilon \))
  - Get extra leading bit for “free”
An Example of a Normalized Float

Value
Float \( F = 15213.0; \)
- \( 15213_{10} = 11101101101101_2 = 1.1101101101101_2 \times 2^{13} \)

Significand
- \( M = 1.1101101101101_2 \)
- \( \frac{\text{frac}}{} = 11011011011010000000000000_2 \)

Exponent
- \( E = 13 \)
- \( \text{Bias} = 127 \)
- \( E = \text{Exp} – \text{Bias} \quad \Rightarrow \quad \text{Exp} = 13 + 127 = 140 = 10001100_2 \)
Denormalized Values

Condition
- \( \exp = 000...0 \)

Value
- Exponent value \( E = -\text{Bias} + 1 \)
- Significand value \( M = 0.xxx...x_2 \)
  - \( xxx...x \): bits of \( \text{frac} \)

Cases
- \( \exp = 000...0, \text{frac} = 000...0 \)
  - Represents value 0
  - Note that have distinct values +0 and −0
- \( \exp = 000...0, \text{frac} \neq 000...0 \)
  - Numbers very close to 0.0
  - Lose precision as get smaller
  - “Gradual underflow”
Special Values

Condition

- \( \exp = 111...1 \)

Cases

- \( \exp = 111...1, \frac{\text{frac}}{\text{frac}} = 000...0 \)
  - Represents value \( \infty \) (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., \( 1.0/0.0 = -1.0/-0.0 = +\infty, \ 1.0/-0.0 = -\infty \)

- \( \exp = 111...1, \frac{\text{frac}}{\text{frac}} \neq 000...0 \)
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., \( \sqrt{-1}, \infty - \infty \)
Operations on Floating Points

- **Summing floating numbers:**
  - It has a relative error, epsilon
    - \((a + b) + c \neq a + (b + c)\)
  - Designing efficient algorithms for computing a faithfully rounded floating-point is challenging
  - Proposed several efficient parallel algorithms for summing \(n\) floating-point numbers, so as to produce a faithfully rounded floating-point representation of the sum. [Michael T. Goodrich, Ahmed Eldawy 2016]
Operations on Floating Points (cont.)

- Comparison is tricky:
  - If involved equality, Due to rounding errors, it demands special measures
  - Next slide shows you how to handle it
Floating Point Comparison

root [0] double capacity = 1.0
(double) 1.0000000

root [1] capacity == 0.8
(double) 0.2000000

root [2] capacity == 0.1
(double) 0.1000000

root [3] double item = 0.1
(double) 0.1000000

root [4] item <= capacity
(bool) false

root [5] delta = item - capacity
(double) 5.5511151e-17

root [6] double epsilon = 1e-6
(double) 1.0000000e-06

root [7] equal = fabs(delta) < epsilon
(bool) true

root [8] item < capacity || equal
(bool) true

root [9] bool equals(double a, double b)
root[10] return fabs(a - b) < 1e-6;

root[12] item < capacity || equals(...)