FLOATING POINT

Spring, 2011
Euiseong Seo
(euiseong@skku.edu)
Review: Integers

- Representation: unsigned and signed
- Conversion, casting
  - Bit representation maintained but reinterpreted
- Expanding, truncating
  - Truncating == \texttt{mod}
- Addition, negation, multiplication, shifting
  - Operations are \texttt{mod} 2^w
- “Ring” properties hold
  - Associative, commutative, distributive, additive 0 and inverse
- Ordering properties do not hold
  - \( u > 0 \) does not mean \( u + v > v \)
  - \( u,v > 0 \) does not mean \( u \cdot v > 0 \)
Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary
What is $1011.101_2$?
**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 Number

<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>0.111111₁₂</td>
</tr>
</tbody>
</table>

★ Observations

- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form 0.111111...₂ just below 1.0
  - $1/2 + 1/4 + 1/8 + ... + 1/2^i + ... \rightarrow 1.0$
  - Use notation $1.0 - \epsilon$
**Representable Numbers**

**Limitation**
- Can only exactly represent numbers of the form \( \frac{x}{2^k} \)
- Other rational numbers have repeating bit representations

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>0.01010101010101010101...2</td>
</tr>
<tr>
<td>1/5</td>
<td>0.00110011001100110011...2</td>
</tr>
<tr>
<td>1/10</td>
<td>0.000110011001100110011...2</td>
</tr>
</tbody>
</table>
TODAY: FLOATING POINT

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary
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 fast in hardware
  - Numerical analysts predominated over hardware designers in defining standard
Floating Point 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 \( s \) is sign bit \( s \)
- \( \text{exp} \) field encodes \( E \) (but is not equal to \( E \))
- \( \text{frac} \) field encodes \( M \) (but is not equal to \( M \))
Precisions

- Single precision: 32 bits
  - Format: \( s \) exp \( \frac{1}{8} \) frac
  - Sign: 1
  - Exponent: 8
  - Fraction: 23

- Double precision: 64 bits
  - Format: \( s \) exp \( \frac{1}{11} \) frac
  - Sign: 1
  - Exponent: 11
  - Fraction: 52

- Extended precision: 80 bits (Intel only)
  - Format: \( s \) exp \( \frac{1}{15} \) frac
  - Sign: 1
  - Exponent: 15
  - Fraction: 63 or 64
**Normalized Values**

- Condition: $\exp \neq 000...0$ and $\exp \neq 111...1$
- Exponent coded as a biased value: $E = \text{Exp} - \text{Bias}$
  - $\text{Exp}$: unsigned value $\exp$
  - $\text{Bias} = 2^{e-1} - 1$, where $e$ is number of exponent bits
    - Single precision: 127 ($\text{Exp} = 1...254$, $E: -126...127$)
    - Double precision: 1023 ($\text{Exp} = 1...2046$, $E: -1022...1023$)
- Significand coded with implied leading 1: $M = 1.xxx...x_2$
  - $xxx...x$: bits of $\text{frac}$
  - Minimum when $000...0$ ($M = 1.0$)
  - Maximum when $111...1$ ($M = 2.0 - \varepsilon$)
  - Get extra leading bit for “free”
**Normalized Encoding Example**

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

- **Significand**
  - $M = 1.1101101101101_2$
  - $\text{frac} = 11011011011010000000000000_2$

- **Exponent**
  - $E = 13$
  - $\text{Bias} = 127$
  - $\text{Exp} = 140 = 10001100_2$

- **Result**
  - $s \quad \text{exp} \quad \text{frac}$
Denormalized Values

- Condition: \( \text{exp} = 000...0 \)
- Exponent value: \( \text{E} = -\text{Bias} + 1 \) (instead of \( \text{E} = 0 - \text{Bias} \))
- Significand coded with implied leading 0: \( \text{M} = 0.xxx...x \)
  - \( xxx...x \): bits of frac

Cases
- \( \text{exp} = 000...0, \frac{}{} = 000...0 \)
  - Represents value 0
  - Note distinct values: +0 and −0 (why?)
- \( \text{exp} = 000...0, \frac{}{} \neq 000...0 \)
  - Numbers very close to 0.0
  - Lose precision as get smaller
  - Equi-spaced
Special Values

- Condition: \( \text{exp} = 111...1 \)
- Case: \( \text{exp} = 111...1, \frac{\text{frac}}{} = 000...0 \)
  - Represents value \( \infty \) (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., \( 1.0/0.0 = -1.0/-0.0 = +\infty, \ 1.0/-0.0 = -\infty \)
- Case: \( \text{exp} = 111...1, \frac{\text{frac}}{} \neq 000...0 \)
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., \( \sqrt{-1}, +\infty, -\infty, +\infty * 0 \)
TODAY: FLOATING POINT

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

8-bit Floating Point Representation
- Sign bit is in the most significant bit
- Next four bits are the exponent, with a bias of 7.
- Last three bits are frac

Same general form as IEEE Format
- normalized, denormalized
- representation of 0, NaN, infinity
**Values Related to the Exponent**

<table>
<thead>
<tr>
<th>Exp</th>
<th>exp</th>
<th>E</th>
<th>(2^E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>-6</td>
<td>1/64</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-6</td>
<td>1/64</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>-5</td>
<td>1/32</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>-4</td>
<td>1/16</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>-3</td>
<td>1/8</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>-2</td>
<td>1/4</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>-1</td>
<td>1/2</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>+1</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>+2</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>+3</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>+4</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>+5</td>
<td>32</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>+6</td>
<td>64</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>+7</td>
<td>128</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>n/a</td>
<td>(inf, NaN)</td>
</tr>
</tbody>
</table>
## Dynamic Range (Positive Only)

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>-6</td>
<td>0</td>
<td>closest to zero</td>
</tr>
<tr>
<td>Denormalized numbers</td>
<td>0</td>
<td>0000</td>
<td>000</td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0000</td>
<td>001</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0000</td>
<td>010</td>
<td>-6</td>
<td>6/8*1/64 = 6/512</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0000</td>
<td>110</td>
<td>-6</td>
<td>7/8*1/64 = 7/512</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0000</td>
<td>111</td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-6</td>
<td>1/8</td>
<td>closest to 1 below</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-6</td>
<td>9/8</td>
<td>closest to 1 above</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
<td>largest norm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>8/8*1 = 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>9/8*1 = 9/8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>10/8*1 = 10/8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>14/8*128 = 224</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>15/8*128 = 240</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>n/a</td>
<td>inf</td>
<td></td>
</tr>
<tr>
<td>Normalized numbers</td>
<td>0</td>
<td>0001</td>
<td>000</td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0001</td>
<td>001</td>
<td>-6</td>
<td>9/8*1/64 = 9/512</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0110</td>
<td>110</td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0110</td>
<td>111</td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0111</td>
<td>000</td>
<td>0</td>
<td>8/8*1 = 1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0111</td>
<td>001</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0111</td>
<td>010</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1110</td>
<td>110</td>
<td>7</td>
<td>14/8*128 = 224</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1110</td>
<td>111</td>
<td>7</td>
<td>15/8*128 = 240</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1111</td>
<td>000</td>
<td>n/a</td>
<td>inf</td>
</tr>
</tbody>
</table>
6-bit IEEE-like format
- $e = 3$ exponent bits
- $f = 2$ fraction bits
- Bias is $2^{3-1} - 1 = 3$

Notice how the distribution gets denser toward zero.
6-bit IEEE-like format

- $e = 3$ exponent bits
- $f = 2$ fraction bits
- Bias is $2^{3-1} - 1 = 3$
Do It Yourself

- Convert $10.4_{10}$ to single precision floating point
- Recall that:

$10.4_{10}$ is $1010.0110_{2}$
1. Normalize
   - $1010.0110_2 \times 2^0 = 1.0100110 \times 2^3$

2. Determine sign bit
   - Positive, so $S = 0$

3. Determine exponent
   - $2^3$ so $3 + \text{bias (}= 127) = 130 = 10000010_2$

4. Determine Significand
   - Drop leading 1 of mantissa, expand to 23 bits = 01001100110011001100110

|   | 10000010 | 01001100110011001100110 |
### Interesting Numbers

<table>
<thead>
<tr>
<th>Description</th>
<th>exp</th>
<th>frac</th>
<th>Numerical</th>
<th>Approx. Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>00...00</td>
<td>00...00</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>Smallest Positive Denormalized</td>
<td>00...00</td>
<td>00...01</td>
<td>$2^{-23,52} \times 2^{-126,1022}$</td>
<td>Single $\approx 1.4 \times 10^{-45}$ Double $\approx 4.9 \times 10^{-324}$</td>
</tr>
<tr>
<td>Largest Denormalized</td>
<td>00...00</td>
<td>11...11</td>
<td>$(1.0 - \varepsilon) \times 2^{-126,1022}$</td>
<td>Single $\approx 1.18 \times 10^{-38}$ Double $\approx 2.2 \times 10^{-308}$</td>
</tr>
<tr>
<td>Smallest Positive Normalized</td>
<td>00...01</td>
<td>00...00</td>
<td>$1.0 \times 2^{-126,1022}$</td>
<td>Just larger than largest denormalized</td>
</tr>
<tr>
<td>One</td>
<td>01...11</td>
<td>00...00</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>Largest Normalized</td>
<td>11...10</td>
<td>11...11</td>
<td>$(2.0 - \varepsilon) \times 2^{127,1023}$</td>
<td>Single $\approx 3.4 \times 10^{38}$ Double $\approx 1.8 \times 10^{308}$</td>
</tr>
</tbody>
</table>
**Special Properties of Encoding**

- FP (Floating Point) zero same as integer zero
  - All bits are zero
- Can (Almost) use unsigned integer comparison
  - Must first compare sign bits
  - Must consider \(-0 = 0\)
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denormalized vs. normalized
    - Normalized vs. infinity
Today: Floating Point

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

- $x +_f y = \text{Round}(x + y)$
- $x \times_f y = \text{Round}(x \times y)$

**Basic idea**
- First compute exact result
- Make it fit into desired precision
  - Possibly overflow if exponent too large
  - Possibly round to fit into $\text{frac}$
# Rounding 4 Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>$1.40</th>
<th>$1.60</th>
<th>$1.50</th>
<th>$2.50</th>
<th>-$1.50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Towards zero</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>-$1</td>
</tr>
<tr>
<td>Round down (-(\infty))</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>-$2</td>
</tr>
<tr>
<td>Round up (+(\infty))</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>$3</td>
<td>-$1</td>
</tr>
<tr>
<td>Nearest Even (default)</td>
<td>$1</td>
<td>$2</td>
<td>$2</td>
<td>$2</td>
<td>-$2</td>
</tr>
</tbody>
</table>

- **Round down**
  - Rounded result is close to but no greater than true result.

- **Round up**
  - Rounded result is close to but no less than true result.

- What are the advantages of the modes?
Closer Look at Round-To-Even

- Default rounding mode
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under-estimated

- Applying to other decimal places / bit positions
  - When exactly halfway between two possible values
    - ROUND SO THAT LEAST SIGNIFICANT DIGIT IS EVEN
  - E.g., round to nearest hundredth
    - 1.2349999 → 1.23 (Less than half way)
    - 1.2350001 → 1.24 (Greater than half way)
    - 1.2350000 → 1.24 (Half way—round up)
    - 1.2450000 → 1.24 (Half way—round down)
## Rounding Binary Numbers

- **Binary Fractional Numbers**
  - “Even” when least significant bit is 0
  - “Half way” when bits to right of rounding position = $100..._2$

- **Examples**
  - Round to nearest 1/4 (2 bits right of binary point)

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Rounded</th>
<th>Action</th>
<th>Rounded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3/32</td>
<td>$10.00011_2$</td>
<td>$10.00_2$</td>
<td>down</td>
<td>2</td>
</tr>
<tr>
<td>2 3/16</td>
<td>$10.00110_2$</td>
<td>$10.01_2$</td>
<td>up</td>
<td>2 1/4</td>
</tr>
<tr>
<td>2 7/8</td>
<td>$10.11100_2$</td>
<td>$11.00_2$</td>
<td>up</td>
<td>3</td>
</tr>
<tr>
<td>2 5/8</td>
<td>$10.10100_2$</td>
<td>$10.10_2$</td>
<td>down</td>
<td>2 1/2</td>
</tr>
</tbody>
</table>
Floating Point Multiplication

Exact Result: \((-1)^{s_1} \times M_1 \times 2^{E_1} \times (-1)^{s_2} \times M_2 \times 2^{E_2}\)

- **Fixing**
  - If \(M \geq 2\), shift \(M\) right, increment \(E\)
  - If \(E\) out of range, overflow
  - Round \(M\) to fit \(\text{frac}\) precision

- **Implementation**
  - Biggest chore is **MULTIPLYING SIGNIFICANDS**
**Floating Point Addition**

\[ (-1)^{s_1} \times M_1 \times 2^{E_1} + (-1)^{s_2} \times M_2 \times 2^{E_2} \]

- **Assume** \( E_1 > E_2 \)
- **Exact Result:** \( (-1)^s \times M \times 2^E \)
  - Sign \( s \), significand \( M \):
    - Result of signed align & add
  - Exponent \( E \): \( E_1 \)
- **Fixing**
  - If \( M \geq 2 \), shift \( M \) right, increment \( E \)
  - \( M < 1 \), shift \( M \) left \( k \) positions, decrement \( E \) by \( k \)
  - Overflow if \( E \) out of range
  - Round \( M \) to fit \( \text{frac} \) precision
Mathematical Properties of FP Add

▶ Compare to those of Abelian Group

- Closed under addition? Yes
  - But may generate infinity or NaN
- Commutative? Yes
- Associative? No
  - Overflow and inexactness of rounding
- 0 is additive identity? Yes
- Every element has additive inverse
  - Except for infinities & NaNs Almost

▶ Monotonicity

- \( a \geq b \Rightarrow a+c \geq b+c? \) Almost
  - Except for infinities & NaNs
Mathematical Properties of FP Mult

► Compare to Commutative Ring
  ◦ Closed under multiplication? Yes
    • But may generate infinity or NaN
  ◦ Multiplication Commutative? Yes
  ◦ Multiplication is Associative? No
    • Possibility of overflow, inexactness of rounding
  ◦ 1 is multiplicative identity? Yes
  ◦ Multiplication distributes over addition? No
    • Possibility of overflow, inexactness of rounding

► Monotonicity
  ◦ $a \geq b \; \& \; c \geq 0 \implies a \times c \geq b \times c?$ Almost
    • Except for infinities & NaNs
Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary
**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 $T_{\text{min}}$
  - `int` ⇒ `double`
    - Exact conversion, as long as `int` has $\leq 53$ bit word size
  - `int` ⇒ `float`
    - Will round according to rounding mode
Floating Point Puzzles

For each of the following C expressions, either:

- Argue that it is true for all argument values
- Explain why not true

```c
int x = ...;
float f = ...;
double d = ...;
```

Assume neither `d` nor `f` is NaN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0 ⇒ ((d*2) < 0.0)`
- `d > f ⇒ -f > -d`
- `d * d >= 0.0`
- `(d + f) - d == f`
Today: Floating Point

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

Represents numbers of form $M \times 2^E$

One 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 & serious numerical applications programmers