Lecture 3B
Floating Point

Topics
- IEEE Floating Point Standard
- Rounding
- Floating Point Operations
- Mathematical properties

---

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

---

Floating Point Puzzles
- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = ...;
floaf x = ...;
double d = ...;
```

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (float) d`
- `f == -(f);`

Assume neither `d` nor `f` is NaN

- `2/3 == 2/3.0`
- `d < 0.0  =>  ((d+2) < 0.0)`
- `d > f  =>  -f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

---

Fractional Binary Numbers

Representation
- Bits to right of “binary point” represent fractional powers of 2
- Represents rational number: $\sum_{k=-j}^{i} b_k \cdot 2^k$
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>
<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 0.1111₁₂ just below 1.0
  - 1/2 + 1/4 + 1/8 + ... + 1/2ⁿ + ... \rightarrow 1.0
  - Use notation 1.0 – ε

Representable Numbers

Limitation
- Can only exactly represent numbers of the form \( x/2^k \)
- Other numbers have repeating bit representations

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>0.0101010101[01]₁₂</td>
</tr>
<tr>
<td>1/5</td>
<td>0.001100110011[0011]₁₂</td>
</tr>
<tr>
<td>1/10</td>
<td>0.0001100110011[0011]₁₂</td>
</tr>
</tbody>
</table>

Floating Point Representation

Numerical Form
- \(-1^s M \times 2^E\)
  - Sign bit \( s \) determines whether number is negative or positive
  - Significant \( M \) normally a fractional value in range [1.0,2.0).
  - Exponent \( E \) weights value by power of two

Encoding
- MSB \( s \) is sign bit
- \( \text{exp} \) field encodes \( E \)
- \( \text{frac} \) field encodes \( M \)

Sizes
- Single precision: 8 \( \text{exp} \) bits, 23 \( \text{frac} \) bits
  - 32 bits total
- Double precision: 11 \( \text{exp} \) bits, 52 \( \text{frac} \) bits
  - 64 bits total
- Extended precision: 15 \( \text{exp} \) bits, 63 \( \text{frac} \) bits
  - Only found in Intel-compatible machines
  - Stored in 80 bits
  - 1 bit wasted
- Quad precision: 15 \( \text{exp} \) bits, 112 \( \text{frac} \) bits
  - Stored in 128 bits
"Normalized" Numeric Values

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

Exponent coded as biased value
- \( E = \exp - \text{Bias} \)
  - \( \exp \): unsigned value denoted by \( \exp \)
  - \( \text{Bias} \): Bias value
    - Single precision: 127 (\( \exp: 1\ldots254, E: -126\ldots-127 \))
    - Double precision: 1023 (\( \exp: 1\ldots2046, E: -1022\ldots-1023 \))
    - Quad (Extended) precision: 16383 (\( \exp: 32766, E: -16382\ldots16383 \))
  - In general: \( \text{Bias} = 2^e - 1 \), where \( e \) is number of exponent bits

Significant coded with implied leading 1
- \( M = 1.xxx\ldots x_2 \)
  - \( xxx\ldots x_2 \): bits of frac
  - Minimum when \( 000\ldots0 \) (\( M = 1.0 \))
  - Maximum when \( 111\ldots1 \) (\( M = 2.0 - e \))
  - Get extra leading bit for "free"

Denormalized Values

Condition
- \( \exp = 000\ldots0 \)

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

Cases
- \( \exp = 000\ldots0, \frac{a}{c} = 000\ldots0 \)
  - Represents value 0
  - Note that have distinct values +0 and -0
- \( \exp = 000\ldots0, \frac{a}{c} \neq 000\ldots0 \)
  - Numbers very close to 0.0
  - Lose precision as get smaller
  - "Gradual underflow"

Special Values

Condition
- \( \exp = 111\ldots1 \)

Cases
- \( \exp = 111\ldots1, \frac{a}{c} = 000\ldots0 \)
  - Represents value \( \infty \) (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., \( 1.0/0.0 = -1.0/-0.0 = +\infty, 1.0/0.0 = -\infty \)
- \( \exp = 111\ldots1, \frac{a}{c} \neq 000\ldots0 \)
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., \( \text{sqrt}(-1), \infty/0 \)
Summary of Floating Point Real Number Encodings

Tiny Floating Point Example

8-bit Floating Point Representation
- the sign bit is in the most significant bit.
- the next four bits are the exponent, with a bias of 7.
- the last three bits are the \textit{frac}

- \textit{Same General Form as IEEE Format}
  - normalized, denormalized
  - representation of 0, NaN, infinity

\begin{align*}
\text{s} & \quad \text{exp} & \quad \text{frac} \\
-1 & \quad 3 & \quad 2 & \quad 0
\end{align*}

Dynamic Range

\begin{tabular}{|c|c|c|c|c|}
\hline
\text{s} & \text{exp} & \text{frac} & \text{E} & \text{Value} \\
\hline
0 & 0000 & 000 & -6 & 0 & \text{closest to zero} \\
0 & 0000 & 001 & -6 & 1/8*1/64 = 1/512 \\
0 & 0000 & 010 & -6 & 2/8*1/64 = 2/512 \\
0 & 0001 & 110 & -6 & 6/8*1/64 = 6/512 & \text{largest denom} \\
0 & 0001 & 111 & -6 & 7/8*1/64 = 7/512 & \text{smallest norm} \\
0 & 001 & 000 & -6 & 8/8*1/64 = 8/512 \\
0 & 001 & 001 & -6 & 9/8*1/64 = 9/512 \\
0 & 010 & 110 & -1 & 14/8*1/2 = 14/16 & \text{closest to 1 below} \\
0 & 010 & 111 & -1 & 15/8*1/2 = 15/16 \\
0 & 011 & 000 & 0 & 8/8*1 = 1 & \text{closest to 1 above} \\
0 & 011 & 001 & 0 & 9/8*1 = 9/8 \\
0 & 011 & 100 & 0 & 10/8*1 = 10/8 \\
0 & 110 & 110 & 7 & 14/8*128 = 224 & \text{largest norm} \\
0 & 110 & 111 & 7 & 15/8*128 = 240 \\
0 & 111 & 000 & n/a & \text{inf} \\
\hline
\end{tabular}

Values Related to the Exponent

\begin{tabular}{|c|c|c|c|}
\hline
\text{Exp} & \text{exp} & \text{E} & 2^E \\
\hline
0 & 0000 & -6 & 1/64 (denorms) \\
1 & 0001 & -6 & 1/64 \\
2 & 0010 & -5 & 1/32 \\
3 & 0011 & -4 & 1/16 \\
4 & 0100 & -3 & 1/8 \\
5 & 0101 & -2 & 1/4 \\
6 & 0110 & -1 & 1/2 \\
7 & 0111 & 0 & 1 \\
8 & 1000 & +1 & 2 \\
9 & 1001 & +2 & 4 \\
10 & 1010 & +3 & 8 \\
11 & 1011 & +4 & 16 \\
12 & 1100 & +5 & 32 \\
13 & 1101 & +6 & 64 \\
14 & 1110 & +7 & 128 \\
15 & 1111 & \text{n/a} & \text{(inf, NaN)} \\
\hline
\end{tabular}
## Distribution of Values

6-bit IEEE-like format
- **e** = 3 exponent bits
- **f** = 2 fraction bits
- Bias is 3

Notice how the distribution gets denser toward zero.

![Distribution of Values](image1)

## 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>00...00 00...00</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>Smallest Pos. Denorm.</td>
<td>00...00 00...01</td>
<td>2^-203'e2 X 2^-126 (0,1022)</td>
<td></td>
</tr>
</tbody>
</table>
  - Single: 1.4 X 10^-26
  - Double: 4.9 X 10^-304
| Largest Denormalized         | 00...00 11...11 | (1.0 - e) X 2^-126 (126,1022) |
  - Single: 1.18 X 10^-30
  - Double: 2.2 X 10^-305
| Smallest Pos. Normalized     | 00...01 00...00 | 1.0 X 2^-126 (126,1022) |
| One                          | 01...11 00...00 | 1.0  |
| Largest Normalized           | 11...10 11...11 | (2.0 - e) X 2^127 (1023) |
  - Single: 3.4 X 10^18
  - Double: 1.8 X 10^309

## Special Properties of Encoding

**FP Zero Same as Integer Zero**
- All bits = 0
- But -0.0 = TMIN

**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
  - Denorm vs. normalized
  - Normalized vs. infinity
Floating Point Operations

Conceptual View
- First compute exact result
- Make it fit into desired precision
  - Possibly overflow if exponent too large
  - Possibly round to fit into \( \text{frac} \)

Rounding Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>1.4</th>
<th>1.6</th>
<th>1.5</th>
<th>2.5</th>
<th>-1.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round 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>Round towards Nearest Even (default)</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>-2</td>
</tr>
</tbody>
</table>

Note:
1. Round down: rounded result is close to but no greater than true result.
2. Round up: rounded result is close to but no less than true result.

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>(&lt;1/2—down)</td>
<td>2</td>
</tr>
<tr>
<td>2 3/16</td>
<td>10.00110_2</td>
<td>10.01_2</td>
<td>(&gt;1/2—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>(1/2—up)</td>
<td>3</td>
</tr>
<tr>
<td>2 5/8</td>
<td>10.10100_2</td>
<td>10.10_2</td>
<td>(1/2—down)</td>
<td>2 1/2</td>
</tr>
</tbody>
</table>

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)

FP Multiplication

Operands
\((-1)^s M_1 x 2^{E_1} \quad \times \quad (-1)^t M_2 x 2^{E_2}\)

Exact Result
\((-1)^{s+t} M \times 2^{E}\)
- Sign: \(s \times t\)
- Significand: \(M_1 \times M_2\)
- Exponent: \(E_1 + E_2\)

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
**FP Addition**

**Operands**

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

**Assume E1 > E2**

**Exact Result**

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

**Sign s, significand M:**
- Result of signed align & add
- Exponent E: E1

**Fixing**

- If M ≥ 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

---

**Floating Point in C**

C Guarantees Two Levels

- `float` single precision
- `double` double precision

Conversions

- Casting between int, float, and double changes numeric values
- Note that casting between int and float changes bit-representation
- Double or float to int
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range
  - Generally saturates to TMin or TMax
- `int` to `double`
  - Exact conversion, as long as int has ≤ 53 bit word size
- `int` to `float`
  - Will round according to rounding mode

---

**Answers to Floating Point Puzzles**

```c
int x = -.1;
float f = -.1;
double d = -.1;
```

- Assume neither d nor f is NaN
- x == (int)(float) x (No: 24 bit significand)
- x == (int)(double) x (Yes: 53 bit significand)
- f == (float)(double) f (Yes: increases precision)
- d == (float) d (No: loses precision)
- f == -(-f); (Yes: Just change sign bit)
- 2/3 == 2/3.0 (No: 2/3 != 0)
- d < 0.0 ⇒ ((d*2) < 0.0) (Yes!)
- d > f ⇒ -f > -d (Yes!)
- d * d >= 0.0 (Yes!)
- (d+f)-d == f (No: Not associative)

---

**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 & serious numerical applications programmers