FLOATING POINT

COMPUTER ARCHITECTURE AND ORGANIZATION
Today: Floating Point

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

What is $1011.101_2$?
Fractional Binary Numbers

- **Representation**
  - Bits to right of “binary point” represent fractional powers of 2
  - Represents rational number: \[ \sum_{k=-j}^{i} b_k \times 2^k \]
## Fractional Binary Numbers: 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>1 7/16</td>
<td>1.0111₁₂</td>
</tr>
<tr>
<td>63/64</td>
<td>0.11111₁₂</td>
</tr>
</tbody>
</table>

### Observations
- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form 0.11111₁₂... are just below 1.0
  - $1/2 + 1/4 + 1/8 + ... + 1/2^i + ... \rightarrow 1.0$
- Use notation $1.0 - \varepsilon$
Representable Numbers

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

- Value
  - 
  - $1/3$ 0.0101010101[01]...$_2$
  - $1/5$ 0.001100110011[0011]...$_2$
  - $1/10$ 0.0001100110011[0011]...$_2$
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:**
  \[ (-1)^s M \times 2^E \]
  - **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 \( s \)
  - **exp field** encodes \( E \) (but is not equal to \( E \))
  - **frac field** encodes \( M \) (but is not equal to \( M \))
Precisions

- **Single precision: 32 bits**
  
  - s
  - 8-bits
  - exp
  - 23-bits
  - frac

- **Double precision: 64 bits**
  
  - s
  - 11-bits
  - exp
  - 52-bits
  - frac

- **Extended precision: 80 bits (Intel only)**
  
  - s
  - 15-bits
  - exp
  - 63 or 64-bits
  - frac
Normallized Values

- Condition: exp $\neq$ 000...0 and exp $\neq$ 111...1

- Exponent coded as *biased* value: $E = Exp - Bias$
  - $Exp$: unsigned value exp
  - $Bias = 2^{k-1} - 1$, where $k$ is number of exponent bits
    - Single precision: 127 (Exp: 1...254, E: -126...127)
    - Double precision: 1023 (Exp: 1...2046, E: -1022...1023)

- Significand coded with implied leading 1: $M = 1.xxx...x_2$
  - $xxx...x$: bits of 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**
  - $M = 1.1101101101101$
  - $\frac{frac}{frac} = 1101101101101000000000000_2$

- **Exponent**
  - $E = 13$
  - $Bias = 127$
  - $Exp = 140 = 10001100_2$

- **Result:**
  - $s 10001100 1101101101101010000000000000$
Denormalized Values

- Condition: \(\text{exp} = 000\ldots0\)
- Exponent value: \(E = \text{-Bias} + 1\) (instead of \(E = 0 - \text{Bias}\))
- Significand coded with implied leading 0: \(M = 0.xxx\ldots x_2\)
  - \(xxx\ldots x\): bits of \(\text{frac}\)
- Cases
  - \(\text{exp} = 000\ldots0, \text{frac} = 000\ldots0\)
    - Represents zero value (why +0 and -0?)
  - \(\text{exp} = 000\ldots0, \text{frac} \neq 000\ldots0\)
    - Numbers very close to 0.0
    - Lose precision as get smaller
    - Equispaced
- \(1.23 \times 10^{-6}\) is normalized, \(0.01 \times 10^{-6}\) is denormalized
  - All +/- of unequal norms have non-zero result (gradual underflow)
Special Values

- Condition: $\text{exp} = 111\ldots1$

- Case: $\text{exp} = 111\ldots1$, $\text{frac} = 000\ldots0$
  - Represents value $\infty$ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$

- Case: $\text{exp} = 111\ldots1$, $\text{frac} \neq 000\ldots0$
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., $\sqrt{-1}$, $\infty - \infty$, $\infty \times 0$
Visualization: Floating Point Encodings
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**
  - 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 *frac*

- **Same general form as IEEE Format**
  - normalized, denormalized
  - representation of 0, NaN, infinity
## Dynamic Range (Positive Only)

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0000</td>
<td>-6</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0001</td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0010</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>110</td>
<td>-6</td>
<td>6/8*1/64 = 6/512</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>111</td>
<td>-6</td>
<td>7/8*1/64 = 7/512</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0001</td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0011</td>
<td>-6</td>
<td>9/8*1/64 = 9/512</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1111</td>
<td>0</td>
<td>8/8*1 = 1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1111</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1111</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1110</td>
<td>7</td>
<td>14/8*128 = 224</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1110</td>
<td>7</td>
<td>15/8*128 = 240</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1111</td>
<td>n/a</td>
<td>inf</td>
</tr>
</tbody>
</table>

**Denormalized numbers**
- Closest to zero
- Largest denorm
- Smallest norm

**Normalized numbers**
- Closest to 1 below
- Closest to 1 above
- Largest norm
Distribution of Values

- 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.
Distribution of Values (close-up view)

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

[Diagram showing distribution of values with labels for denormalized, normalized, and infinity.]
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</td>
<td>00...00</td>
<td>0.0</td>
</tr>
<tr>
<td>Smallest Pos. Denorm.</td>
<td>00...00</td>
<td>00...01</td>
<td>$2^{-{23,52}} \times 2^{-{126,1022}}$</td>
</tr>
<tr>
<td>Single ≈ 1.4 $\times$ $10^{-45}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double ≈ 4.9 $\times$ $10^{-324}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Largest Denormalized</td>
<td>00...00</td>
<td>11...11</td>
<td>$(1.0 - \varepsilon) \times 2^{-{126,1022}}$</td>
</tr>
<tr>
<td>Single ≈ 1.18 $\times$ $10^{-38}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double ≈ 2.2 $\times$ $10^{-308}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smallest Pos. Normalized</td>
<td>00...01</td>
<td>00...00</td>
<td>$1.0 \times 2^{-{126,1022}}$</td>
</tr>
<tr>
<td>Just larger than largest denormalized</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>One</td>
<td>01...11</td>
<td>00...00</td>
<td>1.0</td>
</tr>
<tr>
<td>Largest Normalized</td>
<td>11...10</td>
<td>11...11</td>
<td>$(2.0 - \varepsilon) \times 2^{{127,1023}}$</td>
</tr>
<tr>
<td>Single ≈ 3.4 $\times$ $10^{38}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double ≈ 1.8 $\times$ $10^{308}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Special Properties of Encoding

- **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
    - What should comparison yield?
  - Otherwise OK
    - Denorm 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: Basic Idea

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

- Basic idea
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into frac
## Rounding

- **Rounding Modes (illustrate with $ rounding)**

<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 (-∞)</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>$-2</td>
</tr>
<tr>
<td>Round up (+∞)</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>

- 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>10.00011₂</td>
<td>10.00₂</td>
<td>(&lt;1/2—down)</td>
<td>2</td>
</tr>
<tr>
<td>2 3/16</td>
<td>10.00110₂</td>
<td>10.01₂</td>
<td>(&gt;1/2—up)</td>
<td>2 1/4</td>
</tr>
<tr>
<td>2 7/8</td>
<td>10.11100₂</td>
<td>11.00₂</td>
<td>(1/2—up)</td>
<td>3</td>
</tr>
<tr>
<td>2 5/8</td>
<td>10.10100₂</td>
<td>10.10₂</td>
<td>(1/2—down)</td>
<td>2 1/2</td>
</tr>
</tbody>
</table>
FP Multiplication

- \((-1)^{s_1} M_1 2^{E_1} \times (-1)^{s_2} M_2 2^{E_2}\)
- **Exact Result:** \((-1)^{\cdot} M \cdot 2^E\)
  - Sign \(s\): \(s_1 \land s_2\)
  - Significand \(M\): \(M_1 \times M_2\)
  - Exponent \(E\): \(E_1 + E_2\)

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

- **Implementation**
  - Biggest chore is multiplying significands
Mathematical Properties of FP Add

- Compare to those of Abelian Group
  - Closed under addition? \( \text{Yes} \)
    - But may generate infinity or NaN
  - Commutative? \( \text{Yes} \)
  - Associative? \( \text{No} \)
    - Overflow and inexactness of rounding
  - 0 is additive identity? \( \text{Yes} \)
  - Every element has additive inverse \( \text{Almost} \)
    - Except for infinities & NaNs

- Monotonicity
  - \( a \geq b \Rightarrow a+c \geq b+c ? \) \( \text{Almost} \)
    - 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**
  - $a \geq b \land c \geq 0 \Rightarrow a \ast c \geq b \ast c$? *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 Tmin
  - `int` → `double`
    - Exact conversion, as long as `int` has ≤ 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
Today: Floating Point

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