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

- Value | Representation
  - 5 3/4 | 101.11\_2
  - 2 7/8 | 10.111\_2
  - 1 7/16 | 1.0111\_2

- Observations
  - Divide by 2 by shifting right (unsigned)
  - Multiply by 2 by shifting left
  - Numbers of form 0.111111…\_2 are just below 1.0
    - \(1/2 + 1/4 + 1/8 + \ldots + 1/2^j + \ldots \approx 1.0\)

Representable Numbers

- Limitation #1
  - Can only exactly represent numbers of the form \(s/2^e\)
  - Other rational numbers have repeating bit representations

- Limitation #2
  - Only representable numbers are of the form \(s/2^e\)

- What if the number of bits is limited?
  - "Fixed point": just one setting of binary point within the \(w\) bits
  - Limited range of numbers (bad for very small or very large values)
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
  - A lot of work to make fast in hardware
  - Numerical analysts predominated over hardware designers in defining standard

Floating Point Representation

- Numerical Form:

  \[(\pm 1). M \times 2^E\]

  - 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 \(M\))

Precision options

- Single precision: 32 bits
  - 1 exp
  - 8-bits
  - 23-bits
- Double precision: 64 bits
  - 1 exp
  - 11-bits
  - 52-bits
- Extended precision: 80 bits (older Intel only)
  - 1 exp
  - 15-bits
  - 63 or 64-bits

“Normalized” (Normal) Values

- When: \(exp \neq 000...0\) and \(exp \neq 111...1\)
- Exponent coded as a biased value: \(E = Exp - Bias\)
  - Exp: unsigned value of exp field
  - Bias = \(2^{k} - 1\), 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: \(M = 1.xxx...x\)
  - \(xxx...x\): bits of frac field
  - Minimum when \(frac=000...0\) (\(M = 1.0\))
  - Maximum when \(frac=111...1\) (\(M = 2.0 - \epsilon\))
  - Get extra leading bit for “free”

Normalized Encoding Example

- Value: float \(F = 15213.0\):
  - \(15213_{10} = 11011010110101_{2}\)
  - \(1.110110101101101 \times 2^{15}\)
- Significand
  - \(M = 1.110110101101101_{2}\)
- Exponent
  - \(E = 13\)
- Result:
  - \(0 10001100 \text{ 110110101101000000000000}_{2}\)
Denormalized Values

- **Condition**: exp = 000...0
- **Exponent value**: E = 1 - Bias (instead of E = 0 - Bias)
- **Significand coded with implied leading 0**: M = 0.xxx...x
  - xxx...x: bits of frac
- **Cases**
  - exp = 000.0, frac = 000.0
    - Represents zero value
    - Note distinct values: +0 and −0 (why?)
  - exp = 000.0, frac ≠ 000.0
    - Numbers closest to 0.0
    - Equispaced

Special Values

- **Condition**: exp = 111...1
- **Case**: exp = 111...1, frac = 000...0
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
    - E.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞
- **Case**: exp = 111...1, frac ≠ 000...0
  - Not a Number (NaN)
  - Represents case when no numeric value can be determined
    - E.g., sqrt(-1), ∞ − ∞, ∞ × 0

Visualization: Floating Point Encodings

- Normalized
- Denormal
- NaN

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

Today: Floating Point

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

Dynamic Range (Positive Only)

<table>
<thead>
<tr>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>-6</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>-1</td>
<td>15/8*1/64 = 15/512</td>
</tr>
</tbody>
</table>

Denormalized numbers

<table>
<thead>
<tr>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1110</td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
</tr>
</tbody>
</table>

Normalized numbers

<table>
<thead>
<tr>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>0</td>
<td>8/8*1 = 1</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>7</td>
<td>14/8*128 = 224</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>7</td>
<td>15/8*128 = 240</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>7</td>
<td>16/8*128 = 256</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1111</td>
<td>7</td>
<td>NaN</td>
</tr>
</tbody>
</table>

- closest to zero
- closest to 1 above
- closest to 1 below
- closest to 1
- largest norm
- smallest norm
Distribution of Values

- **6-bit IEEE-like format**
  - \( e = 3 \) exponent bits
  - \( f = 2 \) fraction bits
  - Bias is \( 2^{3-1} = 3 \)

- Notice how the distribution gets denser toward zero.

Special Properties of the IEEE 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

Floating Point Operations: Basic Idea

- \( x +_e y = \text{Round}(x + y) \)

- \( x \times_e y = \text{Round}(x \times y) \)

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

Today: Floating Point

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

Rounding

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

<table>
<thead>
<tr>
<th></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 ( \leftarrow )</td>
<td>$1</td>
<td>$1</td>
<td>$1</td>
<td>$2</td>
<td>$-2</td>
</tr>
<tr>
<td>Round up ( \rightarrow )</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 different modes good for?**
  - Towards zero: compatible with C integer behavior
  - Round down/up: maintain conservative intervals
  - Nearest even: unbiased, minimal error
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

<table>
<thead>
<tr>
<th>Number</th>
<th>Rounded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.8849999</td>
<td>7.88</td>
</tr>
<tr>
<td>7.8950001</td>
<td>7.89</td>
</tr>
<tr>
<td>7.8950000</td>
<td>7.89</td>
</tr>
<tr>
<td>7.8850000</td>
<td>7.88</td>
</tr>
</tbody>
</table>

Exercise break: FP and money?

- Your sandwich shop uses single-precision floating point for sales amounts
- Need to apply a Minneapolis sales tax of 7.75%, rounded up to the nearest cent
- On $4.00 purchase, compute:
  - round_up((4.00 * 0.0775 * 100) = 32 cents
  - Correct tax is 31 cents
- What went wrong?
  - Note: 0.0775 = 31/400 exactly
  - Think about the answer first, then see the choices on ChimeIn: https://chimein.cla.umn.edu/course/view/2021

FP and money: what went wrong?

- 0.0775 = 31/400 cannot be represented exactly in binary
  - 400 is not a power of 2
- Actual representation with be like 0.0775 ± ϵ
  - For single-precision, closest is 0.0775 + ϵ
- 4.00 * (0.775 + ϵ) * 100 = 31 + ϵ
  - round_up((31 + ϵ) = 32
- Similar problems can happen with double precision or other rounding modes
  - Real Minnesota law is a more complex rule
- Better choices:
  - Store cents or smaller fractions as an integer, or
  - Special libraries for decimal arithmetic

FP Multiplication

- (-1)^s1 M1 2^e1 × (-1)^s2 M2 2^e2
- Exact Result: (-1)^s M 2^e
  - Sign s:
    - s1 ^ s2
  - Significand M:
    - M1 x M2
  - Exponent E:
    - E1 + E2

- Fixing
  - If M ≥ 2, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit frac precision

- Implementation
  - Biggest chore is multiplying significands

Floating Point Addition

- (-1)^s1 M1 2^e1 + (-1)^s2 M2 2^e2
  - Assume E1 > E2

- Exact Result: (-1)^s M 2^e
  - Sign s, significand M:
    - Result of signed align & add
  - Exponent E:
    - E1 - E2

- 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

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
    - (3.14 + 1e10) - 1e10 = 0, 3.14 + (1e10 - 1e10) = 3.14
  - 0 is additive identity?
    - Yes
  - Every element has additive inverse?
    - Yes, except for infinities & NaNs
  - Monotonicity
    - a ≥ b ⇒ a + c ≥ 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
    - Ex: (1e20*1e20)*1e20 = inf, 1e20*(1e20*1e20) = 1e20
  - 1 is multiplicative identity?
    - Yes
  - Multiplication distributes over addition?
    - Yes
  - Possibility of overflow, inexactness of rounding
    - 1e20*(1e20*1e20) = 0.0, 1e20*1e20 - 1e20*1e20 = NaN

- **Monotonicity**
  - $a \geq b \& c \geq 0 \Rightarrow a \cdot c \geq b \cdot c$?
    - Almost

---

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 (full)

- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Explain why not true
  
  ```
  int x = ...;
  float f = ...;
  double d = ...;
  Assume neither d nor f is NaN
  ```

  - $x == (int)(float) x$
  - $x == (int)(double) x$
  - $f == (float)(double) f$
  - $d == (double)(float) d$
  - $f == (-f)$
  - $2/3 == 2/3.0$
  - $d < 0.0 \Rightarrow (d*2 < 0.0)$
  - $d > f \Rightarrow -f > -d$
  - $d + d == 0.0$
  - $(d+f)-d == f$

---

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

---

Additional Slides
Creating Floating Point Number

- **Steps**
  - Normalize to have leading 1
  - Round to fit within fraction
  - Postnormalize to deal with effects of rounding

- **Case Study**
  - Convert 8-bit unsigned numbers to tiny floating point format

Example Numbers

- **Value** | **Binary** | **Fraction** | **Exponent**
- 128 | 10000000 | 1.0000000 | 7
- 15 | 00001101 | 1.1010000 | 3
- 17 | 00010001 | 1.0010000 | 4
- 19 | 00010011 | 1.0011000 | 4
- 138 | 10001010 | 1.0001010 | 7
- 63 | 00111111 | 1.1111100 | 5

Normalize

- **Requirement**
  - Set binary point so that numbers of form 1.xxxxx
  - Adjust all to have leading one
  - Decrement exponent as shift left

<table>
<thead>
<tr>
<th>Value</th>
<th>Fraction</th>
<th>GRS</th>
<th>Incr?</th>
<th>Rounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1.0000000</td>
<td>000</td>
<td>N</td>
<td>1.000</td>
</tr>
<tr>
<td>15</td>
<td>1.1010000</td>
<td>100</td>
<td>N</td>
<td>1.101</td>
</tr>
<tr>
<td>17</td>
<td>1.0001000</td>
<td>010</td>
<td>N</td>
<td>1.000</td>
</tr>
<tr>
<td>19</td>
<td>1.0011000</td>
<td>110</td>
<td>Y</td>
<td>1.010</td>
</tr>
<tr>
<td>138</td>
<td>1.0001010</td>
<td>011</td>
<td>Y</td>
<td>1.001</td>
</tr>
<tr>
<td>63</td>
<td>1.1111100</td>
<td>111</td>
<td>Y</td>
<td>1.000</td>
</tr>
</tbody>
</table>

Rounding

- **Guard bit:** LSB of result
- **Sticky bit:** OR of remaining bits

<table>
<thead>
<tr>
<th>Value</th>
<th>Fraction</th>
<th>GRS</th>
<th>Incr?</th>
<th>Rounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1.0000000</td>
<td>000</td>
<td>N</td>
<td>1.000</td>
</tr>
<tr>
<td>15</td>
<td>1.1010000</td>
<td>100</td>
<td>N</td>
<td>1.101</td>
</tr>
<tr>
<td>17</td>
<td>1.0001000</td>
<td>010</td>
<td>N</td>
<td>1.000</td>
</tr>
<tr>
<td>19</td>
<td>1.0011000</td>
<td>110</td>
<td>Y</td>
<td>1.010</td>
</tr>
<tr>
<td>138</td>
<td>1.0001010</td>
<td>011</td>
<td>Y</td>
<td>1.001</td>
</tr>
<tr>
<td>63</td>
<td>1.1111100</td>
<td>111</td>
<td>Y</td>
<td>1.000</td>
</tr>
</tbody>
</table>

Postnormalize

- **Issue**
  - Rounding may have caused overflow
  - Handle by shifting right once & incrementing exponent

<table>
<thead>
<tr>
<th>Value</th>
<th>Rounded</th>
<th>Exp</th>
<th>Adjusted</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1.000</td>
<td>7</td>
<td></td>
<td>128</td>
</tr>
<tr>
<td>15</td>
<td>1.101</td>
<td>3</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>17</td>
<td>1.000</td>
<td>4</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>19</td>
<td>1.010</td>
<td>4</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>138</td>
<td>1.001</td>
<td>7</td>
<td></td>
<td>134</td>
</tr>
<tr>
<td>63</td>
<td>10.000</td>
<td>5</td>
<td>1.000/6</td>
<td>64</td>
</tr>
</tbody>
</table>

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>$2^{-1.53} \times 2 = (2,1022)$</td>
</tr>
<tr>
<td>Largest Denormalized</td>
<td>00...00</td>
<td>11...11</td>
<td>$(1.0-\epsilon) \times 2 = (127,1012)$</td>
</tr>
<tr>
<td>Smallest Pos. Normalized</td>
<td>00...01</td>
<td>00...00</td>
<td>$2^{-1.53}$</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>$(2.0-\epsilon) \times 2^{127,1012}$</td>
</tr>
</tbody>
</table>