Introduction to Computer Systems
15-213/18-243, spring 2009
4th Lecture, Jan. 22nd

Instructors:
Gregory Kesden and Markus Püschel
Last Time: Integers

- Representation: unsigned and signed
- Conversion, casting
  - Bit representation maintained but reinterpreted
- Expanding, truncating
  - Truncating = mod
- Addition, negation, multiplication, shifting
  - Operations are 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
Fractional binary numbers

What is 1011.101?
Fractional Binary Numbers

- 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 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>10.111₁₂</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 $0.\overline{111111}_2$ are 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 $x/2^k$
  - Other rational numbers have repeating bit representations

- **Value**
  - **Value**
  - **Representation**
  - $1/3$: $0.0101010101[01]..._2$
  - $1/5$: $0.001100110011[0011]..._2$
  - $1/10$: $0.0001100110011[0011]..._2$
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:**
  \[ (-1)^s \ M \ 2^E \]
  - **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 exp frac
  - 1 8 23

- Double precision: 64 bits
  - s exp frac
  - 1 11 52

- Extended precision: 80 bits (Intel only)
  - s exp frac
  - 1 15 63 or 64
Normalized Values

- **Condition:** \( \text{exp} \neq 000...0 \) and \( \text{exp} \neq 111...1 \)

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

- **Significand coded with implied leading 1:** \( M = 1 . \text{xxx...x}_2 \)
  - \( \text{xxx...x} \): 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**: \( \text{Float } F = 15213.0; \)
  - \( 15213_{10} = 11101101101101_2 \)
    \[ = 1.1101101101101_2 \times 2^{13} \]

- **Significand**
  - \( M = 1.1101101101101 \)
  - \( \text{frac} = 110110110110100000000000_2 \)

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

- **Result**:
  \[
  \begin{array}{c|c|c}
  s & \text{exp} & \text{frac} \\
  \hline
  0 & 10001100 & 11011011011010000000000000000000 \\
  \end{array}
  \]
Denormalized Values

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

- **Exponent value:** \( E = -\text{Bias} + 1 \) (instead of \( E = 0 - \text{Bias} \))

- **Significand coded with implied leading 0:** \( M = 0 . \text{xxx}...x \)
  - \( \text{xxx}...x \): bits of \( \text{frac} \)

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

- **Condition**: $\exp = 111...1$

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

- **Case**: $\exp = 111...1$, $\frac{\text{frac}}{\text{frac}} \neq 000...0$
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., $\sqrt{-1}$, $\infty - \infty$, $\infty \times 0$
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>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 000</td>
<td>-6</td>
<td>0</td>
<td>0</td>
<td>closest to zero</td>
</tr>
<tr>
<td>0000 001</td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 010</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 110</td>
<td>-6</td>
<td>6/8*1/64 = 6/512</td>
<td>largest denorm</td>
<td></td>
</tr>
<tr>
<td>0000 111</td>
<td>-6</td>
<td>7/8*1/64 = 7/512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001 000</td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
<td>smallest norm</td>
<td></td>
</tr>
<tr>
<td>0001 001</td>
<td>-6</td>
<td>9/8*1/64 = 9/512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 000</td>
<td>0</td>
<td>8/8*1 = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010 001</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
<td>closest to 1 above</td>
<td></td>
</tr>
<tr>
<td>0011 000</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110 000</td>
<td>0</td>
<td>14/8*128 = 224</td>
<td>largest norm</td>
<td></td>
</tr>
<tr>
<td>0110 001</td>
<td>0</td>
<td>15/8*128 = 240</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111 000</td>
<td>n/a</td>
<td>inf</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

- Denormalized
- Normalized
- Infinity
Distribution of Values (close-up view)

6-bit IEEE-like format
- e = 3 exponent bits
- f = 2 fraction bits
- Bias is 3
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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single</td>
<td>$\approx 1.4 \times 10^{-45}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>$\approx 4.9 \times 10^{-324}$</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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single</td>
<td>$\approx 1.18 \times 10^{-38}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>$\approx 2.2 \times 10^{-308}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smallest Pos. Normalized</td>
<td>00...01</td>
<td></td>
<td>00...00 $1.0 \times 2^{-{126,1022}}$</td>
</tr>
<tr>
<td></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>
</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

- $\text{x } +_{f} \text{ y } = \text{Round}(\text{x } + \text{ y})$

- $\text{x } \times_{f} \text{ y } = \text{Round}(\text{x } \times \text{ y})$

**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)

- Towards zero
  - $1.40: $1
  - $1.60: $1
  - $1.50: $1
  - $2.50: $2
  - $–1.50: $–1

- Round down (-∞)
  - $1.40: $1
  - $1.60: $1
  - $1.50: $1
  - $2.50: $2
  - $–1.50: $–2

- Round up (+∞)
  - $1.40: $2
  - $1.60: $2
  - $1.50: $2
  - $2.50: $3
  - $–1.50: $–1

- Nearest Even (default)
  - $1.40: $1
  - $1.60: $2
  - $1.50: $2
  - $2.50: $2
  - $–1.50: $–2

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>010.00011_2</td>
<td>010.00_2</td>
<td>(&lt;1/2—down)</td>
<td>2</td>
</tr>
<tr>
<td>2 3/16</td>
<td>010.00110_2</td>
<td>010.01_2</td>
<td>(&gt;1/2—up)</td>
<td>2 1/4</td>
</tr>
<tr>
<td>2 7/8</td>
<td>010.11100_2</td>
<td>011.00_2</td>
<td>( 1/2—up)</td>
<td>3</td>
</tr>
<tr>
<td>2 5/8</td>
<td>010.10100_2</td>
<td>010.10_2</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:** \((-1)^s M \ 2^E\)

- 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 \(\text{frac}\) precision

**Implementation**

- Biggest chore is multiplying significands
Floating Point Addition

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

- **Exact Result:** \( (-1)^s \ M \ 2^E \)
  - Sign \( s \), significand \( M \):
    - Result of signed align & add
  - Exponent \( E \): \( E_1 \)

- **Fixing**
  - If \( M \geq 2 \), shift \( M \) right, increment \( E \)
  - If \( M < 1 \), 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 **Almost**
    - Except for infinities & NaNs

- **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**
  - \(a \geq b \& c \geq 0 \Rightarrow a \times c \geq b \times c?\) **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 TMin
  - `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

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

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
More 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
  - 128 10000000
  - 15  00001101
  - 33  00010001
  - 35  00010011
  - 138 10001010
  - 63  00111111
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>Binary</th>
<th>Fraction</th>
<th>Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>10000000</td>
<td>1.0000000</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>00001101</td>
<td>1.1010000</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>00010001</td>
<td>1.0001000</td>
<td>5</td>
</tr>
<tr>
<td>19</td>
<td>00010011</td>
<td>1.0011000</td>
<td>5</td>
</tr>
<tr>
<td>138</td>
<td>10001010</td>
<td>1.0001010</td>
<td>7</td>
</tr>
<tr>
<td>63</td>
<td>00111111</td>
<td>1.1111100</td>
<td>5</td>
</tr>
</tbody>
</table>
## Rounding

### Round up conditions

- Round = 1, Sticky = 1 $\Rightarrow$ > 0.5
- Guard = 1, Round = 1, Sticky = 0 $\Rightarrow$ Round to even

<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>1.00000000</td>
<td>000</td>
<td>N</td>
<td>1.000</td>
</tr>
<tr>
<td>15</td>
<td>1.10100000</td>
<td>100</td>
<td>N</td>
<td>1.101</td>
</tr>
<tr>
<td>17</td>
<td>1.00010000</td>
<td>010</td>
<td>N</td>
<td>1.000</td>
</tr>
<tr>
<td>19</td>
<td>1.00110000</td>
<td>110</td>
<td>Y</td>
<td>1.010</td>
</tr>
<tr>
<td>138</td>
<td>1.00010100</td>
<td>011</td>
<td>Y</td>
<td>1.001</td>
</tr>
<tr>
<td>63</td>
<td>1.11111000</td>
<td>111</td>
<td>Y</td>
<td>10.000</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>