Floating Point Arithmetic

September 28, 2000

Topics

- IEEE Floating Point Standard
- Rounding
- Floating Point Operations
- Mathematical properties
- IA32 floating point
Floating Point Puzzles

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

```plaintext
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`
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
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 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>

Observation

- Divide by 2 by shifting right
- Numbers of form \(0.111111\ldots₂\) just below 1.0
  - Use notation 1.0 – ε

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]\ldots₂</td>
</tr>
<tr>
<td>1/5</td>
<td>0.001100110011[0011]\ldots₂</td>
</tr>
<tr>
<td>1/10</td>
<td>0.0001100110011[0011]\ldots₂</td>
</tr>
</tbody>
</table>
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 is sign bit
- exp field encodes \( E \)
- frac field encodes \( M \)

Sizes

- Single precision: 8 exp bits, 23 frac bits
  - 32 bits total
- Double precision: 11 exp bits, 52 frac bits
  - 64 bits total
“Normalized” Numeric Values

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

Exponent coded as biased value
\[ E = \text{Exp} - \text{Bias} \]
– \( \text{Exp} \) : unsigned value denoted by \( \exp \)
– \( \text{Bias} \) : Bias value
  » Single precision: 127 (\( \text{Exp}: 1\ldots254, \ E: -126\ldots127 \))
  » Double precision: 1023 (\( \text{Exp}: 1\ldots2046, \ E: -1022\ldots1023 \))
  » in general: \( \text{Bias} = 2^{m-1} - 1 \), where \( m \) is the number of exponent bits

Significand coded with implied leading 1
\[ m = 1.\text{xxx}\ldots\text{x}_2 \]
– \( \text{xxx}\ldots\text{x} \): bits of \( \text{frac} \)
– Minimum when \( 000\ldots0 \) (\( M = 1.0 \))
– Maximum when \( 111\ldots1 \) (\( M = 2.0 - \varepsilon \))
– 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

\[ M = \begin{array}{c}
1.11011011011012 \\
frac = \begin{array}{c}
1101101101101000000000002
\end{array}
\end{array} \]

Exponent

\[ E = 13 \]
\[ Bias = 127 \]
\[ Exp = 140 = 10001100_{2} \]

Floating Point Representation (Class 02):

<table>
<thead>
<tr>
<th>Hex:</th>
<th>4</th>
<th>6</th>
<th>6</th>
<th>D</th>
<th>B</th>
<th>4</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary:</td>
<td>0100</td>
<td>0110</td>
<td>0110</td>
<td>1101</td>
<td>1011</td>
<td>0100</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>140:</td>
<td>100</td>
<td>0110</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15213:</td>
<td>1110</td>
<td>1101</td>
<td>1011</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Denormalized Values

Condition

• \( \exp = 000...0 \)

Value

• Exponent value \( E = -Bias + 1 \)
• Significand value \( m = 0.\text{xxx...x}_2 \)
  \(-\text{xxx...x}: \text{bits of frac}\)

Cases

• \( \exp = 000...0, \frac{}{} = 000...0 \)
  – Represents value 0
  – Note that have distinct values +0 and –0

• \( \exp = 000...0, \frac{}{} \neq 000...0 \)
  – Numbers very close to 0.0
  – Lose precision as get smaller
  – “Gradual underflow”
## Interesting Numbers

<table>
<thead>
<tr>
<th>Description</th>
<th>$\text{exp}$</th>
<th>$\text{frac}$</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>$\cdot$ Single $\approx 1.4 \times 10^{-45}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\cdot$ 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>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\cdot$ Single $\approx 1.18 \times 10^{-38}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\cdot$ Double $\approx 2.2 \times 10^{-308}$</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></td>
<td></td>
<td></td>
<td>$\cdot$ Just larger than largest denormalized</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></td>
<td></td>
<td></td>
<td>$\cdot$ Single $\approx 3.4 \times 10^{38}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\cdot$ Double $\approx 1.8 \times 10^{308}$</td>
</tr>
</tbody>
</table>
Special Values

Condition

• \( \text{exp} = 111...1 \)

Cases

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

• \( \text{exp} = 111...1, \text{frac} \neq 000...0 \)
  – Not-a-Number (NaN)
  – Represents case when no numeric value can be determined
  – E.g., sqrt(−1), \( \infty - \infty \)
Summary of Floating Point Real Number Encodings

-∞ - Normalized

-∞ - Denorm

0 +0 + Denorm

+∞ + Normalized

NaN

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)

```
    7 6 3 2 0
     |   |   |
     S  exp frac
```
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

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>000</td>
<td>-6</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>001</td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>010</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>110</td>
<td>-6</td>
<td>6/8*1/64 = 6/512</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>111</td>
<td>-6</td>
<td>7/8*1/64 = 7/512</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>000</td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>001</td>
<td>-6</td>
<td>9/8*9/64 = 9/512</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>110</td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>111</td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>000</td>
<td>0</td>
<td>8/8*1 = 1</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>001</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>010</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>110</td>
<td>7</td>
<td>14/8*128 = 224</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>111</td>
<td>7</td>
<td>15/8*128 = 240</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>000</td>
<td>n/a</td>
<td>inf</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
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 frac

Rounding Modes (illustrate with $ 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>Zero</td>
<td>$1.00</td>
<td>$1.00</td>
<td>$1.00</td>
<td>$2.00</td>
<td>−$1.00</td>
</tr>
<tr>
<td>Round down (−∞)</td>
<td>$1.00</td>
<td>$1.00</td>
<td>$1.00</td>
<td>$2.00</td>
<td>−$2.00</td>
</tr>
<tr>
<td>Round up (+∞)</td>
<td>$2.00</td>
<td>$2.00</td>
<td>$2.00</td>
<td>$3.00</td>
<td>−$1.00</td>
</tr>
<tr>
<td>Nearest Even (default)</td>
<td>$1.00</td>
<td>$2.00</td>
<td>$2.00</td>
<td>$2.00</td>
<td>−$2.00</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.
A 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

• 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

Operands

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

Exact Result

\[( -1)^s \ M \ 2^E\]

- Sign \( s \): \( s_1 \wedge s_2 \)
- 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
FP Addition

Operands

\((-1)^s_1 M_1 \ 2^{E_1}\)
\((-1)^s_2 M_2 \ 2^{E_2}\)

• 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 frac precision
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

Montonicity

• \( a \geq b \Rightarrow a+c \geq b+c \) ? \( \text{ALMOST} \)
  – Except for infinities & NaNs
Algebraic 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

Montonicity

• $a \geq b$ & $c \geq 0 \Rightarrow a \times c \geq b \times c$? ALMOST
  – Except for infinities & NaNs
Floating Point in C

C Guarantees Two Levels

float single precision
double double precision

Conversions

• Casting between int, float, and double changes numeric values
  • 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 ≤ 54 bit word size
  • int to float
    – Will round according to rounding mode
Answers to Floating Point Puzzles

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

Assume neither 
d nor f is NAN

- \( x == (\text{int})(\text{float}) x \):
  No: 24 bit significand

- \( x == (\text{int})(\text{double}) x \):
  Yes: 53 bit significand

- \( f == (\text{float})(\text{double}) f \):
  Yes: increases precision

- \( d == (\text{float}) d \):
  No: loses precision

- \( f == -(-f); \):
  Yes: Just change sign bit

- \( 2/3 == 2/3.0 \):
  No: 2/3 == 1

- \( d < 0.0 \Rightarrow ((d*2) < 0.0) \):
  Yes!

- \( d > f \Rightarrow -f < -d \):
  Yes!

- \( d * d >= 0.0 \):
  Yes!

- \( (d+f)-d == f \):
  No: Not associative
IA32 Floating Point

History
- 8086: first computer to implement IEEE FP
  - separate 8087 FPU (floating point unit)
- 486: merged FPU and Integer Unit onto one chip

Summary
- Hardware to add, multiply, and divide
- Floating point data registers
- Various control & status registers

Floating Point Formats
- single precision (C `float`): 32 bits
- double precision (C `double`): 64 bits
- extended precision (C `long double`): 80 bits
FPU Data Register Stack

FPU register format (extended precision)

<table>
<thead>
<tr>
<th></th>
<th>79</th>
<th>78</th>
<th>64</th>
<th>63</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>s</td>
<td></td>
<td>exp</td>
<td>frac</td>
</tr>
</tbody>
</table>

FPU register stack

- stack grows down
  - wraps around from R0 -> R7
- FPU registers are typically referenced relative to top of stack
  - st(0) is top of stack (Top)
  - followed by st(1), st(2), ...
- push: increment Top, load
- pop: store, decrement Top
FPU instructions

Large number of floating point instructions and formats

- ~50 basic instruction types
- load, store, add, multiply
- sin, cos, tan, arctan, and log!

Sampling of instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fldz</td>
<td>push 0.0</td>
<td>Load zero</td>
</tr>
<tr>
<td>flds S</td>
<td>push S</td>
<td>Load single precision real</td>
</tr>
<tr>
<td>fmuls S</td>
<td>st(0) &lt;- st(0)*S</td>
<td>Multiply</td>
</tr>
<tr>
<td>faddp</td>
<td>st(1) &lt;- st(0)+st(1); pop</td>
<td>Add and pop</td>
</tr>
</tbody>
</table>
Floating Point Code Example

Compute Inner Product of Two Vectors

- Single precision arithmetic
- Scientific computing and signal processing workhorse

```c
float ipf (float x[],
          float y[],
          int n)
{
    int i;
    float result = 0.0;
    for (i = 0; i < n; i++) {
        result += x[i] * y[i];
    }
    return result;
}
```

```
pushl %ebp              # setup
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx       # %ebx=&x
    movl 12(%ebp),%ecx      # %ecx=&y
    movl 16(%ebp),%edx      # %edx=n
    fldz                    # push +0.0
    xorl %eax,%eax          # i=0
    cmpl %edx,%eax          # if i>=n done
    jge .L3
    .L5:
        flds (%ebx,%eax,4)     # push x[i]
        fmuls (%ecx,%eax,4)    # st(0)*=y[i]
        faddp                   # st(1)+=st(0); pop
        incl %eax               # i++
        cmp %edx,%eax          # if i<n repeat
        jge .L3
        movl -4(%ebp),%ebx     # finish
        leave
        ret                     # st(0) = result
```
Inner product stack trace

1. fldz
   st(0) 0

2. flds (%ebx,%eax,4)
   st(0) x[0]
   st(1) 0

3. fmuls (%ecx,%eax,4)
   st(0) x[0]*y[0]
   st(1) 0

4. faddp %st,%st(1)
   st(0) 0 + x[0]*y[0]

5. flds (%ebx,%eax,4)
   st(0) x[1]
   st(1) 0 + x[0]*y[0]

6. fmuls (%ecx,%eax,4)
   st(0) x[0]*y[0]
   st(1) 0 + x[0]*y[0]

7. faddp %st,%st(1)
   st(0) 0 + x[0]*y[0] + x[1]*y[1]
Summary

IEEE Floating Point Has Clear Mathematical Properties

- Represents numbers of form \( x \times 2^n \)
- 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

IA32 Floating Point is a Mess

- Ill-conceived, pseudo-stack architecture
- Covered in notes