Floating Point Puzzles

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

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

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

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 0.111111₂ 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 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]₂</td>
</tr>
<tr>
<td>1/5</td>
<td>0.001100110011[0011]₂</td>
</tr>
<tr>
<td>1/10</td>
<td>0.0001100110011[0011]₂</td>
</tr>
</tbody>
</table>

### Floating Point Representation

**Numerical Form**
- $-1^s \frac{M}{2^E}$
  - 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 $M$

### Floating Point Precisions

**Sizes**
- Single precision: 8 exp bits, 23 frac bits
  - 32 bits total
- Double precision: 11 exp bits, 52 frac bits
  - 64 bits total
- Extended precision: 15 exp bits, 63 frac bits
  - Only found in Intel-compatible machines
  - Stored in 80 bits
- 1 bit wasted
“Normalized” Numeric Values

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

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

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
- \( \text{Float } F = 15213.0; \)
  - \( 15213_{10} = 1.11011011011012 \times 2^{13} \)

Significant
- \( M = 1.11011011011012 \)
- \( \text{frac} = 1110110110110100000000000_2 \)

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

Floating Point Representation:

<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 0110 0110 1101 1011 0100 0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>140:</td>
<td>100 0110 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15213:</td>
<td>1110 1101 1011 01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Denormalized Values

Condition
- \( \text{exp} = 000...0 \)

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

Cases
- \( \text{exp} = 000...0, \text{frac} = 000...0 \)
  - Represents value 0
  - Note that have distinct values +0 and −0
- \( \text{exp} = 000...0, \text{frac} \neq 000...0 \)
  - Numbers very close to 0.0
  - Lose precision as get smaller
  - “Gradual underflow”

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., \( \text{sqrt}(-1) \), \( \infty - \infty \), \( \infty \times 0 \)
Summary of Floating Point Real Number Encodings

-∞ - Normalized - Denorm + Denorm + Normalized + ∞

NaN

-∞ - Denorm +0 +0 + Normalized - ∞

NaN

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 frac

- Same General Form as IEEE Format
- normalized, denormalized
- representation of 0, NaN, infinity

<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 closest to zero</td>
</tr>
<tr>
<td>0 0000 010 -6 2/8*1/64 = 2/512 Denormalized numbers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>0000 111 -6 6/8*1/64 = 6/512 largest denom</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0001 000 -6 8/8*1/64 = 8/512 smallest norm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0001 001 -6 9/8*1/64 = 9/512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>0 0110 110 -1 14/8*1/2 = 14/16 closest to 1 below</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0110 111 -1 15/8*1/2 = 15/16 closest to 1 above</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0111 000 0 8/8*1 = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0111 001 0 9/8*1 = 9/8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0111 010 0 10/8*1 = 10/8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1110 110 7 14/8*128 = 224 largest norm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1110 111 7 15/8*128 = 240</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1111 000 n/a inf</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dynamic Range

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 (denoms)</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>
Distribution of Values

6-bit IEEE-like format
- e = 3 exponent bits
- f = 2 fraction bits
- Bias is 3
- ± 14 largest representable finite numbers.

Notice how the distribution gets denser toward zero.

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>Largest Denormalized</td>
<td>00...00</td>
<td>11...11</td>
<td>$(1.0 - \epsilon) \times 2^{-126,1022}$</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>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>$(2.0 - \epsilon) \times 2^{127,1023}$</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
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 $\frac{1}{2}$

Rounding Modes (illustrate with $\frac{1}{2}$ rounding)

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
<th>Rounded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1.40$</td>
<td>$1.40$</td>
<td>$1.50$</td>
</tr>
<tr>
<td>$1.60$</td>
<td>$1.60$</td>
<td>$2.50$</td>
</tr>
<tr>
<td>$2.50$</td>
<td>$2.50$</td>
<td>$1.50$</td>
</tr>
</tbody>
</table>

- **Zero**
- **Round down ($-\infty$)**
- **Round up ($+\infty$)**
- **Nearest Even (default)**

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.

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>Value</th>
<th>Rounded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1.2349999$</td>
<td>$1.23$</td>
</tr>
<tr>
<td>$1.2350001$</td>
<td>$1.24$</td>
</tr>
<tr>
<td>$1.2350000$</td>
<td>$1.24$</td>
</tr>
</tbody>
</table>

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>Action</th>
<th>Rounded Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2 \frac{3}{32}$</td>
<td>$10.00011_2$</td>
<td>$&lt;\frac{1}{2}$—down</td>
<td>$2$</td>
</tr>
<tr>
<td>$2 \frac{3}{16}$</td>
<td>$10.00110_2$</td>
<td>$&gt;\frac{1}{2}$—up</td>
<td>$2 \frac{1}{4}$</td>
</tr>
<tr>
<td>$2 \frac{7}{8}$</td>
<td>$10.11100_2$</td>
<td>$\frac{1}{2}$—up</td>
<td>$3$</td>
</tr>
<tr>
<td>$2 \frac{5}{8}$</td>
<td>$10.10100_2$</td>
<td>$&lt;\frac{1}{2}$—down</td>
<td>$2 \frac{1}{2}$</td>
</tr>
</tbody>
</table>

FP Multiplication

Operands

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

Exact Result

$(-1)^s M \cdot 2^E$
- **Sign s:** $s_1 \wedge s_2$
- **Significand M:** $M_1 \cdot M_2$
- **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 $\frac{1}{2}$ precision

Implementation
- Biggest chore is multiplying significands
**FP Addition**

Operands

\[ (-1)^{s1} M_1 \ 2^{E_1} \]

\[ (-1)^{s2} M_2 \ 2^{E_2} \]

- 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 \( \frac{\text{precision}}{2} \)

---

**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

---

**Math. Properties of FP Mult**

**Compare to Commutative Ring**

- Closed under multiplication? \( \text{YES} \)
  - But may generate infinity or NaN
- Multiplication Commutative? \( \text{YES} \)
- Multiplication is Associative? \( \text{NO} \)
  - Possibility of overflow, inexactness of rounding
- 1 is multiplicative identity? \( \text{YES} \)
- Multiplication distributes over addition? \( \text{NO} \)
  - Possibility of overflow, inexactness of rounding

**Monotonicity**

- \( a \geq b \& c \geq 0 \Rightarrow a \cdot c \geq b \cdot c \)? \( \text{ALMOST} \)
  - Except for infinities & NaNs

---

**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**

<table>
<thead>
<tr>
<th>Number</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>10000000</td>
</tr>
<tr>
<td>15</td>
<td>00001101</td>
</tr>
<tr>
<td>33</td>
<td>00010001</td>
</tr>
<tr>
<td>35</td>
<td>00010011</td>
</tr>
<tr>
<td>138</td>
<td>10001010</td>
</tr>
<tr>
<td>63</td>
<td>00111111</td>
</tr>
</tbody>
</table>

---
**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>1.111100</td>
<td>5</td>
</tr>
</tbody>
</table>

**Rounding**

### Round up conditions
- Round = 1, Sticky = 1 ➞ > 0.5
- Guard = 1, Round = 1, Sticky = 0 ➞ 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>1.0000000</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>111</td>
<td>Y</td>
<td>1.001</td>
</tr>
<tr>
<td>63</td>
<td>1.1111100</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>

**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 or NaN
    - Generally sets to Tmin
- int to double
  - Exact conversion, as long as int has ≤ 53 bit word size
- int to float
  - Will round according to rounding mode
Curious Excel Behavior

<table>
<thead>
<tr>
<th>Number Subtract 16</th>
<th>Subtract .3</th>
<th>Subtract .01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Format</td>
<td>16.31</td>
<td>0.31</td>
</tr>
<tr>
<td>Currency Format</td>
<td>$16.31</td>
<td>$0.31</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Spreadsheets use floating point for all computations
- Some imprecision for decimal arithmetic
- Can yield nonintuitive results to an accountant!

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

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

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