On the Absence of Tie-Breaking and Rounding Overflow in Floating-Point Division

Gordon Lichtstein

Problem 1: Tie-Breaking Case in Floating-Point Division

Prove that in floating-point division with inputs and output having the same number of mantissa bits n, the tie-breaking case (where the exact result is halfway between two representable numbers) cannot arise.

Definitions and Framework

1. A floating-point number is represented as $x = M \cdot 2^e$, where M is a sequence of digits $M = 1.m_1m_2m_3 \dots m_n$, where $m_i \in \{0, 1\}$, and n is the number of mantissa bits.

2. Floating-point division amounts to subtracting the exponents, dividing the mantissas using a fixed-point division algorithm, and normalizing the resulting mantissa.

3. Since mantissas implicitly begin with 1 per the floating-point standard, normalization shifts the mantissa left until the leftmost nth digit is a 1. In division, if the numerator's mantissa is smaller than the denominator's, we must shift left by one digit (i.e., multiply by two).

4. The tie-breaking case occurs when the resulting normalized mantissa M is exactly halfway between two representable numbers. That is, the mantissa must be in the form $M = 1.m_1m_2m_3 \dots m_n1 = 1.m_{1,n}1$.

Proof

1. Let $a = A \cdot 2^{e_1}$ and $b = B \cdot 2^{e_2}$ be the inputs, where A, B represent the mantissas in the form $1.a_1a_2a_3 \dots a_n$ and $1.b_1b_2b_3 \dots b_n$ respectively.

2. The exact result of $\frac{a}{b}$ is $\frac{A}{B} \cdot 2^{e_1 - e_2} = M \cdot 2^e$, where M is the resulting (not yet normalized) mantissa.

3. Thus, $M = A/B$, with $0.5 < M < 2$.

4. Observe that $\frac{A}{B} = \frac{2^n \cdot A}{2^n \cdot B}$. So we will now assign $A' := 2^n \cdot A = 1a_1a_2a_3 \ldots a_n = 1a_{1,n}$ and B' respectively, to shift away the decimal points for ease of use.

5. Now we must handle normalization, which splits this proof into two cases.

6. Now for sake of contradiction, assume that the normalized form M_n is unrepresentable, and thus in the form $1.m_1m_2m_3 \ldots m_n1 = 1.m_{1,n}1$.

7. **Case 1:** $A < B$, so $0.5 < M = A/B < 1$

   (a) Then, M is in the form $0.1m_1m_2m_3 \ldots$, so the normalized $M_n = 2M = 1.m_1m_2m_3 \ldots$.

   (b) Assign $M' := 2^n \cdot M_n = 2^{n+1} \cdot M = 1m_{1,n}.1$.

   (c) Now $2^{n+1} \cdot \frac{A'}{B'} = M'$, so $2^{n+1} \cdot A' = M' \cdot B'$. Note that the rightmost $n+1$ (and therefore n) digits of $2^{n+1} \cdot A'$ are 0, and it is an integer.

   (d) Expanding yields $2^{n+1} \cdot A' = 1m_{1,n}.1 \cdot 1b_{1,n} = (1m_{1,n} + 0.1) \cdot 1b_{1,n}$.

   (e) $(1m_{1,n} + 0.1) \cdot 1b_{1,n} = 1m_{1,n} \cdot 1b_{1,n} + 0.1 \cdot 1b_{1,n} = 1m_{1,n} \cdot 1b_{1,n} + 1b_{1,n-1}.b_n$.

   (f) As $2^{n+1} \cdot A'$ is an integer, $b_n = 0$.

   (g) Now consider the units digit (the one directly to the left of the decimal point). As the rightmost n digits of $2^{n+1} \cdot A'$ are 0, it must equal zero. Therefore, $b_n \cdot m_n + b_{n-1} = 0$. However, we already know $b_n = 0$, so $b_{n-1} = 0$.

   (h) We now wish to prove that all $b_i = 0$ for $i \in [1, n)$. Allow our inductive hypothesis to be $p(i) := b_i = 0$. We have proven $b_{n-1} = 0$, so our base case $p(n-1)$ is true.

   (i) Now our (strong) inductive step is proving that $p(i) \ldots p(n-1) \implies p(i-1)$.

   (j) We know that the i^{th} digit of $2^{n+1} \cdot A' = 0$ for $i \in [1, n)$, so

   $$0 = (b_nc_i + b_{n-1}c_{i+1} + \cdots + b_{i+1}c_{n-1} + b_ic_n) + b_{i-1}.$$

   (k) By our proof in step (g), $b_n = 0$, and by the inductive hypothesis, $b_{n-1}, \ldots, b_i = 0$. So

   $$0 = (0 + 0 + \cdots + 0) + b_{i-1} \implies b_{i-1} = 0,$$

   which proves our inductive step.

   (l) Now it is safe to assume all $b_i = 0$ for $i \in [1, n)$. Let's look at what happens to the leading 1 digit in B'.

   (m) We know that the rightmost n digits of $2^{n+1} \cdot A'$ are 0, so

   $$0 = (b_1c_n + b_2c_{n-1} + \cdots + b_nc_1) + 1.$$

   But we know that $b_i = 0$ for $i \in [1, n)$, so

   $$0 = 1 \implies \text{Contradiction.}$$

(n) So assuming A/B for $A < B$ is unrepresentable in floating point leads to a contradiction, so it must be representable.

8. **Case 2:** $A \geq B$, so $1 \leq M = A/B < 2$

   (a) Then, M is in the form $1.m_1m_2m_3\ldots$, so the normalized $M_n = M = 1.m_1m_2m_3\ldots$.

   (b) Assign $M' := 2^n \cdot M_n = 2^n \cdot M = 1m_{1,n}.1$.

   (c) Now $2^n \cdot \frac{A'}{B'} = M'$, so $2^n \cdot A' = M' \cdot B'$.

   (d) Notably, the rest of Case 1 applies, as there is no point in the proof that we rely on the $n+1$th digit from the right being zero. Thus, assuming A/B is unrepresentable for $A \geq B$ also leads to a contradiction.

9. Since the two cases above are exhaustive and both lead to contradictions, A/B must be representable.

1. Since the two cases above are exhaustive and both lead to contradictions, A/B must be representable.

Conclusion

The tie-breaking case is impossible in floating-point division when both inputs and the output use the same number of mantissa bits and are normalized. However, this conclusion does not hold in the presence of subnormal (denormalized) numbers—those whose mantissas do not begin with an implicit leading 1. In such cases, the assumptions underpinning the proof no longer apply. Specifically, the inductive argument in the proof relies on the normalization invariant and breaks down when rightmost $n+1$ bits of the mantissa may not be zero, which is possible when the inputs are non-normal. Therefore, if denormalized numbers are permitted, tie-breaking scenarios cannot be ruled out, and tie-breaking checks must still be implemented in floating-point division logic.

Problem 2: Rounding Cannot Cause Overflow in Floating-Point Division

Prove that in normalized floating-point division, rounding can never cause overflow. If the result overflows, it must have already done so before rounding.

Sources of Overflow in Floating Point Division

Overflow in floating-point division could theoretically arise from the following sources:

1. **Exponent Overflow During Exponent Subtraction:**

   In floating-point division, the exponent of the result is computed as the difference of the input exponents:

   $$e_{\text{out}} = e_1 - e_2.$$

   If this subtraction results in a value greater than the maximum allowed exponent $E_{\max}$, then overflow occurs immediately. This is a pre-rounding overflow and is trivially detected at the exponent computation stage, prior to any mantissa manipulation or rounding.

2. **Overflow During Mantissa Normalization After Division:**

   After dividing the mantissas,

   $$M = A/B,$$

   the result may need to be normalized. If the quotient M is less than 1, it must be shifted left by one bit to restore the implicit leading 1 in the normalized significand, and the exponent is incremented by 1 to compensate.

   This renormalization step might risk causing overflow by increasing the exponent. We observe that:

   - The normalization shift only occurs when $A < B$, so $M = A/B < 1$.
   - After normalization, the result becomes $M_{\text{norm}} = 2M < 2$, and the final value is $M_{\text{norm}} \cdot 2^{e_1 - e_2 - 1 + 1} = 2M \cdot 2^{e_1 - e_2}$.
   - That is, the normalized result is strictly less than $2 \cdot 2^{e_1 - e_2} = 2^{e_1 - e_2 + 1}$.
   - Since the original inputs A and B are representable, and the exponent $e_1 - e_2$ is in range, this result remains bounded and cannot overflow unless the exponent difference itself was already at or above the maximum allowed value.

   Thus, any overflow could be quickly and efficiently detected in exponent calculation. If the resulting exponent calculation is exactly at its maximum value, we check whether $A < B$ and if so, overflow is detected.

3. **Overflow from Rounding a Nearly-Maximal Quotient:**

   One might imagine that rounding could cause overflow if the exact quotient $M = A/B$ lies just below a power of two, specifically in the interval $M \in (1.111\ldots1_2,\ 10.000\ldots0_2)$, where the lower bound is the binary value $1 + (1 - 2^{-n}) = 1.111\ldots1_2$ with $n+1$ ones after the point, which rounds up to $2 = 10.000\ldots0_2$.

   However, this case cannot occur in floating-point division of normalized inputs. The precise bounds for A and B are:

$$A, B \in \left[1,\ 2 - 2^{-n}\right],$$

   corresponding to binary manitssas of the form $1.000\ldots0_2$ (a 1 followed by n zeros) up to $1.111\ldots1_2$ (a 1 followed by n ones). which implies:

$$\frac{A}{B} \in \left[\frac{1}{2 - 2^{-n}},\ 2 - 2^{-n}\right].$$

   The maximum value A/B can reach is:

$$2 - 2^{-n} = 1.111\ldots1_2 \quad (n \text{ ones}),$$

   which is strictly less than:

$$1.111\ldots11_2 \quad (n+1 \text{ ones}),$$

   the smallest value that would round up to $10.000\ldots0_2$.

   Therefore, the quotient A/B can never fall into the critical rounding interval, and rounding cannot cause overflow. If overflow occurs, it must already be present in the unrounded exponent.

Conclusion

Overflow in floating-point division can arise only from the exponent computation or normalization when the exponent is already at its maximum value. In all such cases, the overflow is detectable based on the exponent alone—before any rounding occurs.

Rounding, on the other hand, cannot cause overflow: the quotient A/B resulting from division of normalized mantissas never enters the critical rounding interval that would push it to the next power of two. As shown, the maximum possible quotient A/B is strictly less than the value required to round up to an overflowing result.

Therefore, overflow can always be preemptively detected by inspecting the exponent calculation and, if necessary, comparing A and B when the exponent is at the edge. Rounding does not introduce new overflow; it can only confirm overflow that was already present.