Reduced Precision Checking to Detect Errors in Floating Point Arithmetic
Yaqi Zhang, Ralph Nathan, and Daniel J. Sorin
Duke University

Abstract

In this paper, we use reduced precision checking (RPC) to detect errors in floating point arithmetic. Prior work explored RPC for addition and multiplication. In this work, we extend RPC to a complete floating point unit (FPU), including division and square root, and we present precise analyses of the errors undetectable with RPC that show bounds that are smaller than prior work. We implement RPC for a complete FPU in RTL and experimentally evaluate its error coverage and cost.

1 INTRODUCTION

In this paper, we focus on detecting errors in floating point arithmetic performed by the floating point units (FPUs) in processor cores. We focus on all of the common floating point operations: addition/subtraction, multiplication, division, and square root.

We base our error detection on the previously introduced idea of reduced precision checking (RPC) [1][9]. The key idea behind RPC is to check a full-precision (32-bit, in this paper) FPU with a reduced-precision checker FPU. The full-precision FPU operates on IEEE standard operands that have a sign bit, 8 exponent bits, and 23 fraction bits. The checker FPU operates on operands that also have a sign bit and 8 exponent bits, but the fractions have \( k \) bits, where \( k \leq 23 \) is a parameter that is chosen by the designer.

In the absence of errors, the full-precision result of the FPU will match the reduced-precision result of the checker; only errors in the least significant bits can evade detection. RPC has an inherent trade-off between the cost of the checker and the likelihood of it not detecting an error.

Prior work has developed RPC for addition/subtraction [1] and multiplication [9]. The prior work introduced the idea of RPC and built and evaluated hardware implementations for addition/subtraction and multiplication. Prior work is limited, however, in several important ways. First, it does not include floating point division or square root, both of which are common hardware units. Second, we show that the size of the undetectable errors for addition/subtraction and multiplication can be greatly reduced; prior work made overly conservative assumptions that led to unnecessarily large errors evading detection. Third, for some floating point input operations on some standard operands\(^1\) (e.g., subtraction of two similarly sized numbers), the prior RPC schemes were unable to detect any errors at all and had to be disabled for these operation/operand combinations.

In this paper, we make the following contributions:

- We develop and evaluate RPC for a complete FPU in RTL—addition/subtraction, multiplication, division, and square root. The results include error detection coverage, chip area, and energy.

- We mathematically compute the maximum undetectable error for every floating point operation, and we show that we can obtain tighter bounds for addition/subtraction and multiplication than prior work.

- Using a new RPC technique, that we call “reverse checking,” we eliminate the unchecked operation/operand pairs of prior work. That is, our RPC unit checks every operation on all standard operands.

2 BACKGROUND AND NOTATION

In this section, we introduce background and notation that we use throughout the rest of this paper. We assume that all full-precision floating point numbers adhere to the IEEE-754 standard. Furthermore, we consider only 32-bit floating point numbers in this paper, but our approach applies in an identical way to 64-bit floating point numbers.

2.1 Floating Point Format

As illustrated in Figure 1, a 32-bit floating point number \( X \) has 1 sign bit (\( X[31] \)), 8 exponent bits (\( X[30:23] \)), and 23 fraction bits (\( X[22:0] \)). For notational clarity, we use the symbol \( \exists \) to denote the exponent bits (i.e., \( X[30:23] \)). The sign bit with value of 0 or 1 indicates a positive or negative sign \( (S_X) \), respectively, i.e., \( S_X = (-1)^{X[31]} \). The base-2 exponent is represented in unsigned, biased notation, such that the exponent of \( X \) \( (E_X) \) is the unsigned value of the exponent bits after 127 is subtracted (i.e., \( E_X = X[22:0] - 127 \)). The mantissa of \( X \) is an implicit 1 followed by the 23 fraction bits.\(^2\) We define fraction \( (f_X) \) as the integer value of the lower 23 bits of \( X \) \( (X[22:0]) \), and mantissa \( (M_X) \) as the floating point value of the mantissa, including the implicit one, i.e., \( 1 \cdot f_X \). Curly braces denote concatenation. Thus the floating point value of \( X \) is:

\[
(-1)^{X[31]} \times 2^{X[22:0] - 127} \times 1 \cdot X[22:0]
\]

\[
= S_X \times 2^{E_X} \times 1 \cdot f_X
\]

\[
= S_X \times 2^{E_X} \times M_X
\]

\(^1\) A non-standard operand is a denorm, infinity, or NaN.

\(^2\) In the IEEE format, the “implicit 1” is a 1 at the beginning of the mantissa that is not explicitly stored as part of the 32-bit representation.


2.2 RPC Notation

Because RPC involves computations on portions of a floating point number’s representation, it is helpful to introduce some notation for describing these quantities. Assume that the RPC checker uses a fraction with \( k \) bits, where \( k \leq 23 \). Thus, an RPC checker operates on numbers that have \( 9+k \) bits, where the 9 corresponds to 1 sign bit and 8 exponent bits.

We use \( A \) and \( B \) to refer to the FPU’s operands and \( C \) to refer to its (rounded) result. We use \( \bar{C} \) to denote the “true” result that would be produced with unlimited precision. For example, for multiplication, \( A \times B = \bar{C} \). We use the notation \( A \times B = C \) to denote that \( C \) is the result produced by the hardware. We denote the rounding error associated with the mantissa of \( C (M_C) \) as \( \delta_c \). Thus, \( A \times B = \bar{C} = S_C \times 2^{\varepsilon_C} \times (M_C + \delta_c) \).

Assume the full-precision FPU computes \( A \times B \rightarrow C \), where \( A \), \( B \), and \( C \) are full 32-bit numbers. For purposes of comparing its result, \( C \), to the result of the checker, we consider the most significant \( 9+k \) bits of \( C \) and denote them as \( C' \).

The checker takes as inputs the \( 9+k \) most significant bits of \( A \) and \( B \), which we denote as \( A'^{hi} \) and \( B'^{hi} \), respectively. It produces a \( 9+k \)-bit result that we denote as \( C' \); that is, \( A'^{hi} \times B'^{hi} \rightarrow C' \). To detect errors, RPC compares \( C' \) to \( C' \).

The checker’s inputs, \( A'^{hi} \) and \( B'^{hi} \), are the same as \( A \) and \( B \), except their fractions are truncated. We denote the floating point value of \( A'^{hi} \)'s mantissa, including the implicit one, as \( M'^{hi}_A \). To denote the floating point value of the mantissa that gets truncated off, we use \( M'^{hi}_A \), which has value of \( M_A - M'^{hi}_A \). Because \( M_A \) and \( M'^{hi}_A \) both contain the implicit one, their difference \( M'^{hi}_A \) does not. For instance, if \( M_A = 1.1_{123} \) and \( k = 7 \), \( M'^{hi}_A = 1.1_{17} \) and \( M'^{hi}_A = 0.0_{17} \). The subscript indicates number of repetition of the bit within braces. In this case, \( 0.0_{17} \) means 7 zeros and 16 ones following the decimal point.

The checker’s output, \( C' \), has a fraction and mantissa that we denote as \( f'_{C} \) and \( M'^{hi}_{C} \), respectively.

3 RPC and Its Challenges

At a high level, RPC is similar to dual modular redundancy. Assume the FPU performs a full-precision floating point operation on operands \( A \) and \( B \), say \( A \times B \), and obtains the full-precision result \( C \). To check this result, the checker performs the same floating point operation on the inputs \( A'^{hi} \) and \( B'^{hi} \), where \( A'^{hi} \) and \( B'^{hi} \) are \( 9+k \) most significant bits of operands \( A \) and \( B \), respectively. The checking logic compares the \( 9+k \)-bit result from the checker, \( C' \) to \( A'^{hi} \times B'^{hi} \), to the \( 9+k \) most significant bits of \( C \) from the full-precision FPU, denoted \( C'^{hi} \); if \( C' \) significantly differs from \( C'^{hi} \), an error is reported. We illustrate this simplified version of RPC in Figure 2.

Although RPC is quite simple at this high level of abstraction, it is challenging to efficiently compare \( C'^{hi} \) to \( C' \). With identical dual modular redundancy, comparison between the two results can be efficiently implemented with bitwise comparison. In contrast, we need to compute the difference between the two results, because \( C'^{hi} \) and \( C' \) can differ even in the absence of errors. However, we do not want to perform floating point subtraction, because of the time and energy required to do so.

Instead of comparing the floating point values \( C'^{hi} \) and \( C' \) by performing a floating point subtraction, RPC compares them using \textit{integer} subtraction. First, RPC checks whether the first bits of \( C'^{hi} \) and \( C' \) are equal; these bits are the signs of \( C \) and \( C' \) and they should be equal in the absence of errors. Next, RPC treats the remaining bits of \( C'^{hi} \) and \( C' \) as unsigned integers and computes their integer difference \((C'^{hi}[30:(23-k)] - C'[7+k:0])\), which we refer to as \textit{Diff}. The comparison logic checks whether \textit{Diff} is within a range—bordered by an upper bound (UB) and a lower bound (LB)—that is possible in the absence of errors.

![Figure 1. Representing a floating point number X](image)

![Figure 2. RPC: Big Picture](image)

\[
\text{Diff} = C[30: (23 - k)] - C'[7 + k: 0]
\]

\[
\text{NoError} = (C[31] = C'[31]) \land (LB < \text{Diff} < UB)
\]

In Sections 5-7, we prove that, in the absence of errors:

The bounds on \textit{Diff} for addition and subtraction are \([-1,1]\) and the \([-1,3]\) for multiplication, division, and square root.

If \textit{Diff} is outside of its allowable range, then either an error has occurred in the full-precision FPU or in the checker. (The latter scenario is a false positive.) If \textit{Diff} is non-zero but within its allowable range, then an error may or may not have occurred; if an error did occur, RPC would not detect it.

Later, we prove that the integer values of the bounds on \textit{Diff} do not vary with \( k \). However, the \textit{significance} of these integer bounds does vary. The least significant bit of \textit{Diff} is in the least significant bit position of the checker, which has the floating point value \( 2^{-k} \). Intuitively, a larger value of \( k \) corresponds to a smaller significance of \textit{Diff} (i.e., \( 2^k \) is smaller) and thus a smaller magnitude of error that will not be detected by RPC. As a result, RPC provides a tune-able tradeoff between error coverage and cost.

\[
S_C = (-1)^{\text{sign bit} (1 \text{ bit})} \times E_C = \text{[3-127]} \times (\text{exponent bits (8 bits) fraction (not including implicit 1) (23 bits)})
\]

\[
X^* = \text{[sign bit][exponent bits][first k bits of fraction]}
\]

\[
M^* = 1.0 \times \text{[most significant fraction bits]}
\]

\[
M = 0.0 \times \text{[least significant fraction bits]}
\]
We devote much of this paper to proving how large the integer difference of \( C \) and \( C' \) may be in the absence of errors; any difference larger than this allowable discrepancy will be detected as an error, and any difference smaller will be ignored.

Although integer subtraction is cheap and easy to implement, it introduces the following challenges in bounding the maximum allowable difference in the absence of errors.

### 3.1 Challenge #1: Unequal “Matches” of Exponents

It might appear at first that \( C' \) and \( C \) can differ only in their fractions. However, they can also differ in their exponents in rare situations that must be considered. When we truncate the inputs to the checker (i.e., convert \( A \) and \( B \) into \( A^H \) and \( B^H \) in multiplication, for example), it leads to the absolute value of \( C' \) being less than or equal to the absolute value of \( C \). Thus, the exponent of \( C' (E'_C) \) can be less than the exponent of \( C^H (E_C) \). The mis-match in exponents further causes mis-alignment between the fractions of \( C' \) and \( C \) for purposes of computing integer difference. This rare scenario leads to complexity in bounding \( \text{Diff} \).

### 3.2 Challenge #2: The Need for “Reverse” Checking

The second challenge with RPC is the existence of certain scenarios in which direct checking is ineffective. We illustrate this challenge with a base-10 example: consider the case of \( 3.9999 \times 10^{-3} \) and \( 3.9999 \times 10^{-4} \). When we truncate the inputs to the checker (i.e., convert \( A \) and \( B \) into \( A^H \) and \( B^H \) in multiplication, for example), it leads to the absolute value of \( C' \) being less than or equal to the absolute value of \( C \). Thus, the exponent of \( C' (E'_C) \) can be less than the exponent of \( C^H (E_C) \). The mis-match in exponents further causes mis-alignment between the fractions of \( C' \) and \( C \) for purposes of computing integer difference. This rare scenario leads to complexity in bounding \( \text{Diff} \).

### 3.3 Challenge #3: the Rounding Effect

Another challenge in bounding \( \text{Diff} \) is that both the full-precision FPU and the reduced-precision checker produce inexact, rounded results. We must consider this rounding in two scenarios.

First, rounding in the full-precision unit might produce a carry from low-order bits that could propagate to high-order bits, leading to a mis-match between the high-order bits of \( C' \) and \( C \).

Second, rounding occurs twice in reverse checking, because one input operand used by the checker is rounded (and the result is rounded). For instance, in the case of \( A + B \to C \), even with full-precision \( C \times B \neq A \). With reverse checking, we compute \( C^H \times B^H \to A' \), during which we first round the input \( C^H \) (by truncating \( C \)) and then round the result to get \( A' \).

### 4 Axioms

In the following sections, we frequently use several axioms that derive from the format of IEEE floating point numbers.

- The smallest possible value of \( M_x \) is \( 1.0 \times 2^{-23} = 1 \). The largest possible value of \( M_x \) is \( 1.0 \times 2^{-23} = 1 \).

**Axiom 1:**

\[ 1.0 \times 2^{-23} \leq M_x \leq 2 - 2^{-23} \]

Similarly, the smallest possible value of \( M_x^H \) is also 1 and its largest possible value is \( 1.0 \times 2^{-23} \).

**Axiom 2:**

\[ 1.0 \times 2^{-23} \leq M_x^H \leq 2 - 2^{-23} \]

For \( M_x^H \), the smallest possible value is \( 1.0 \times 2^{-23} \) and the largest possible value is \( 1.0 \times 2^{-23} \).

**Axiom 3:**

\[ 0 \leq M_x^H \leq 2 - 2^{-23} \]

For all floating point operations, the result gets rounded even in the full-precision unit. Take multiplication \( A \times B \to C \) for example:

\[ A \times B = \hat{C} = S_C \times 2^E_C \times (M_C + \delta_C) \quad \text{Eqn. (1)} \]

Rounding error \( \delta_C \) is bounded by machine epsilon (\( \epsilon \)). In the following derivations, we assume the default IEEE rounding mode round-to-nearest (to even on tie), in which case \( \epsilon \) equals \( 2^{-24} \). For the other three modes (toward 0, + \( \infty \), and - \( \infty \)), \( \epsilon \) equals \( 2^{-23} \).

**Axiom 4:**

\[ -2^{-24} \leq \delta_C \leq 2^{-24} \]

For the reduced precision checker, the rounding error (\( \delta_C^H \)) is still 0.5 ulp of the checker. Because the least significant bit in the checker’s mantissa is \( 2^{-k} \), \( \delta_C^H \) is bounded by

**Axiom 5:**

\[ -2^{-k-1} \leq \delta_C^H \leq 2^{-k-1} \]

Note that Axiom 4 and Axiom 5 apply to all operation types, not just multiplication, because we considered only the relationship between the result produced by a FPU and the true result.
result with unlimited precision. Regardless of the type of operation, rounding is performed in the same manner.

An important part of bounding Diff is to connect the integer subtraction used to calculate Diff to the floating point values of its two operands. Consider the integer subtraction for computing Diff in forward checking.

\[
Diff = C[30; (23 - k)]^\text{int} C'[7 + k] \times 0
\]

When \( E_c = E'_c \rightarrow \exists e = \exists \_c \) (common case scenario):

\[
Diff = f_c^\text{int} f'_c
\]

Because 1. \( \{f'_c\} = M'_c \) and 1. \( \{f'_c\} = M'_c \), the difference above equals the floating point difference between \( M'_c \) and \( M'_c \) left shifted by \( k \) (i.e., converted to an integer).

**Axiom 6:** When \( E_c = E'_c \), \( Diff = f_c^\text{int} f'_c \)

Axiom 6 applies to reverse checking similarly, except that, instead of \( C' - C \), \( A' \) or \( B' \) is subtracted from \( A'H \) or \( B'H \), respectively.

## 5 RPC FOR MULTIPLICATION

The full-precision FPU computes \( A \times B \rightarrow C \):

\[
(S_A \times 2^{E_A} \times M_A) \times (S_B \times 2^{E_B} \times M_B) = S_C \times 2^{E_C} \times (M_C + \delta_C)
\]

During error-free execution, \( S_C = S_A \times S_B \). So the signs on both sides of the equation cancel and leave:

\[
M_A \times M_B \times 2^{E_A+E_B} = (M_C + \delta_C) \times 2^{E_C}
\]

\[
M'_C = M_A \times M_B \times 2^{E_A+E_B-\delta_C - M'_C} - \delta'_C
\]

The RPC unit computes \( A'H \times B'H \rightarrow C' \):

\[
(S_A \times 2^{E_A} \times M_A^H) \times (S_B \times 2^{E_B} \times M_B^H) = S_C \times 2^{E_C} \times (M_C + \delta_C)
\]

Because \( S_A \times S_B = S'_C \),

\[
M_H \times M_B^H \times 2^{E_A+E_B} \times (M_C + \delta_C) \times 2^{E_C} \]

\[
M'_C = M_A \times M_B \times 2^{E_A+E_B-\delta_C} - \delta'_C
\]

After the hardware adds \( E_A \) and \( E_B \) and multiplies \( M_A \) and \( M_B \) with integer operations, it must normalize the result. To normalize, the hardware shifts \( M_A \times M_B \) until only one non-zero bit is to the left of the binary point. The number of right shifts is then added to \( E_B \) to get \( E_C \). By Axiom 1, both \( M_A \) and \( M_B \) are less than 2. Therefore \( M_A \times M_B \) is strictly less than 4 (100 in binary), and there can be at most 2 non-zero bits to the left of the binary point. Thus, \( M_A \times M_B \) is right shifted by at most one bit.

In multiplication, the exponent of the product is equal to or is one larger than the sum of the exponents of the operands.

We apply this rule to both the FPU and the checker:

**FPU:** \( E_A + E_B = E_C \) or \( E_A + E_B + 1 = E_C \)

**Checker:** \( E_A + E_B = E'_C \) or \( E_A + E_B + 1 = E'_C \)

Because operands used by the checker are truncated from operands used by the full-precision FPU, the absolute value of \( C' \) is less than or equal to the absolute value of \( C \). So \( E_C \) must be less than or equal to \( E_C \).

\[
E'_C \leq E_C
\]

Combining the conditions above, there are three cases we need to consider.

### 5.1 Common Case 1

Using Eqn. (6), we simplify Eqn. (10) to

\[
M_H^H - M'_C = [M' - M'_C] + [\delta_C - \delta'_C]
\]

Using Axiom 2 and Axiom 3, we can bound boxed term 1 in Eqn. (11), which we denote \( <1> \) hereafter:

\[
0 \leq <1> \leq (2 - 2^{-k}) (2^{-k} - 2^{-23}) \times 2 + (2^{-2} - 2^{-3})^2
\]

\[
= 4 \times (2^{-k} - 2^{-23}) - 2^{-2k} + 2^{-46}
\]

\[
\leq 4 \times (2^{-k} - 2^{-23})
\]

Using Axiom 3, we bound \( <2> \):

\[
-2^{-k} + 2^{-23} < -M'_C \leq 0
\]

Using Axiom 4 and Axiom 5, we bound \( <3> \):

\[
-2^{-3} - 2^{-k-1} \leq \delta_C - \delta'_C \leq 2^{-24} + 2^{-k-1}
\]

We calculate the upper bound of Diff by summing the upper bounds of the three terms and multiplying by \( 2^k \).

\[
Diff = (M_H^H - M'_C) \times 2^k
\]

\[
\leq [4 \times (2^{-k} - 2^{-23}) + 2^{-3} + 2^{-k-1}] \times 2^k < 4.5
\]

Similarly for the lower bound on Diff, we have:

\[
Diff = (M_H^H - M'_C) \times 2^k
\]

\[
\geq [-2^{-3} + 2^{-23} - 2^{-24} - 2^{-k-1}] \times 2^k > -1.5
\]

The above analysis for the upper bound is simplified for purposes of providing intuition. With more sophisticated analysis (Appendix C), we can prove that the upper bound of Diff is strictly less than 4. The reason is because the three bounded terms above are not independent from each other and cannot reach their maximums at the same time. As Diff is an integer, the allowable range of values for Diff in this case is \([-1,3]\).

### 5.2 Common Case 2

Using Eqn. (7), Eqn. (10) can be rewritten as:

\[
M_H^H - M'_C = [M' - M'_C] + [\delta_C - \delta'_C]
\]

Terms \( <2> \) and \( <3> \) are the same as in Section 5.1 and thus have the same bounds. Term \( <1> \) has an additional \( 2^{-1} \) factor not present in Section 5.1. Thus \( <1> \) is bounded by

\[
0 \leq <1> \leq 2 \times (2^{-k} - 2^{-23})
\]

The upper bound on Diff is:

\[
Diff = (M_H^H - M'_C) \times 2^k
\]

\[
\leq [2 \times (2^{-k} - 2^{-23}) + 2^{-24} + 2^{-k-1}] \times 2^k \leq 2.5
\]
The lower bound is the same as in Section 5.1 (i.e., -1). The allowable range for Diff in this case is [-1, 2].

**Summary:** In this section, Appendix B, and Appendix C, we have proved that the allowable range of Diff for multiplication is [-1, 3].

Prior work [9] proved a looser bound of [-7, 7].

6 RPC FOR DIVISION AND SQUARE ROOT

RPC for division and square root uses reverse checking with multiplication. We use the same reduced-precision multiplier that checks multiplication.

6.1 Division

To check \( A + B \rightarrow C \), the checker performs \( C^H \times B^H \rightarrow A' \). In this section, we show that reverse checking of division has the same bounds on Diff as multiplication.

The full precision FPU computes \( A + B \rightarrow C \):

\[
(S_A \times 2^{E_A} \times M_A) + (S_B \times 2^{E_B} \times M_B) = S_C \times 2^{E_C} \times (M_C + \delta_C)
\]

Because \( S_A \times S_B = S_C \),

\[
M_A + M_B \times 2^{E_B-E_A} = (M_C + \delta_C) \times 2^{E_C}
\]

\[
M_A^H = (M_C + \delta_C) \times 2^{E_C} - M_{A'}
\]  
\[
\text{Eqn. (12)}
\]

The checker computes \( C^H \times B^H \rightarrow A' \):

\[
(S_C \times 2^{E_C} \times M_C^H) + (S_B \times 2^{E_B} \times M_B^H) = S_A' \times 2^{E_A} \times (M_A' + \delta_A')
\]

Because \( S_C \times S_B = S_A' \),

\[
M_C^H + M_B^H \times 2^{E_B-E_C} = (M_A' + \delta_A') \times 2^{E_A'}
\]

\[
M_A' = M_C^H M_B^H \times 2^{E_B+E_C-E_A'} - \delta_A'
\]  
\[
\text{Eqn. (13)}
\]

Using similar analysis as for multiplication, because \( 1 \leq M_A, M_B < 2 \) according to Axiom 1, the FPU's quotient \( M_A + M_B \) is in the range of \((0.5, 2)\), open interval. Therefore, the quotient needs to be left-shifted by at most 1 to be normalized such that one bit is to the left of the binary point.

In division, the exponent of the quotient is equal to or is one less than the difference between the exponents of the dividend and divisor.

Applying this rule to the FPU and retaining the same rule for the checker that we had for multiplication, we have:

**FPU:**  \( E_A - E_B = E_C \) or \( E_A - E_B = 1 = E_C \)

**Checker:**  \( E_C + E_B = E_A' \) or \( E_C + E_B = 1 = E_A' \)

The absolute value of \( A' \) is less than or equal to the absolute value of \( A \) due to truncating the mantissas of \( M_C \) and \( M_B \), and thus \( E_A' \leq E_A \).

Applying these conditions above, there are again three cases we must consider:

**Common Case 1:** \( E_B + E_C = E_A' = E_A \)  
\[
\text{Eqn. (14)}
\]

**Common Case 2:** \( E_B + E_C = E_A' + 1 = E_A \)  
\[
\text{Eqn. (15)}
\]

**Corner Case:** \( E_B + E_C + 1 = E_A' + 1 = E_A \)  
\[
\text{Eqn. (16)}
\]

The first two cases are presented in Sections 6.1.1 and 6.1.2. Because \( E_A' = E_A \), Diff = \( (M_A^H - M_{A'}) \times 2^x \) according to Axiom 6. The difference between Eqn. (12) and (13) can be simplified to:

\[
M_A^H - M_{A'} = (M_C^H M_B^H + M_B^H M_C^H + M_B^H M_B^H) \times 2^{E_C+E_B-E_A}-M_{A'} + (\delta_C M_B + 2^{E_C+E_B-E_A} - \delta_A)
\]

Let \( M^* = M_C^H M_B^H + M_B^H M_C^H + M_B^H M_B^H \),

\[
M_A^H - M_{A'} = M^* \times 2^{E_C+E_B-E_A} - M_{A'} + \delta_C M_B \times 2^{E_C+E_B-E_A} - \delta_A
\]  
\[
\text{Eqn. (17)}
\]

The corner case when \( E_A \neq E_A' \) can be analyzed similarly to the corner case of multiplication in Appendix B.

6.1.1 Common Case 1

Using Eqn. (14), Eqn. (17) can be simplified to:

\[
M_A^H - M_A' = \left[ M^* \times 2^{-1} \right]_2 - \left[ M_{A'} \right]_2 + \delta_C M_B \times 2^{-1}
\]  
\[
\text{Eqn. (18)}
\]

\(<1> \) and \(<2> \) have the same boundary conditions as \(<1> \) and \(<2> \) in Section 5.1, respectively. The only difference between Eqn. (18) and Eqn. (11) is that in \(<3> \) \( \delta_C \rightarrow \delta_C \) becomes \( M_B \delta_C + \delta_A' \). Given \( M_B, \delta_C, \) and \( \delta_A' \) are in the range \([1, 2), [-2^{-k-1}, 2^{-k-1}], \) according to Axiom 1, Axiom 4, and Axiom 5, respectively, \(<3> \) now has the following bounds:

\[
-2^{-k-1} - 2^{-2k} < M_B \delta_C + \delta_A' < 2^{-k-1} + 2^{-2k}
\]

which slightly differs from the bounds on \(<3> \) in Section 5.1. This slight difference in this term does not affect the overall bounds on \( M_A^H - M_A' \) as compared to Section 5.1.

\[
\text{Diff} = (M_A^H - M_A') \times 2^k
\]

\[
\leq \left[ 4 \cdot 2^{-2k} + 2^{-2k} + 2^{-k-1} \right] \times 2^k < 4.5
\]

\[
\text{Diff} = (M_A^H - M_A') \times 2^k
\]

\[
\geq [-2^{-k-1} - 2^{-2k} - 2^{-k} + 2^{-2k}] \times 2^k = -1.5
\]

Again, the upper bound can never reach 4 for the same reason as in multiplication. Therefore, the allowable range of Diff is still [-1, 3].

6.1.2 Common Case 2

Using Eqn. (15), Eqn. (17) can be simplified to:

\[
M_A^H - M_A' = \left[ M^* \times 2^{-1} \right]_2 - \left[ M_{A'} \right]_2 + \delta_C M_B \times 2^{-1}
\]

The boundary condition of \(<3> \) becomes

\[
-2^{-k-1} - 2^{-2k} \leq \left( \delta_A + M_B \delta_C \times 2^{-1} \right) \leq 2^{-k} + 2^{-2k}
\]

\(<1> \), \(<2> \), and \(<3> \) have the same boundary conditions as \(<1> \), \(<2> \), and \(<3> \) in Section 5.2, respectively. Therefore, both the upper and lower bounds of Diff are identical to those in Section 5.2.

**Summary:** The corner case of division can be proved in the same manner as multiplication presented in Appendix B. Overall, Diff has the same bounded range for division as for multiplication, which is [-1, 3].

6.2 Square Root

Checking square root is almost identical to checking division. To check \( \sqrt{B} \rightarrow C \), the checker performs \( C^H \times C^H \rightarrow B' \). The analysis for square root is identical to the analysis for division except the two operands of the checker are the same. Therefore, the allowable range of Diff for square root is also [-1, 3].

7 RPC FOR ADDITION/SUBTRACTION

Because of the cancellation problem we discussed in Section 3.2 for addition and subtraction [1], we must separately consider the following two scenarios:

- addition of same-sign operands (SSADD) or subtraction of different-sign operands (DSSUB) (Section 7.1)
• addition of different-sign operands (DSADD) or subtraction of same-sign operands (SSSUB) (Section 7.2)

7.1 Same-Sign Addition (SSADD) or Different-Sign Subtraction (DSSUB)

For both addition of same-sign operands \((S_A = S_B)\) and subtraction of different-sign operands \((S_A \neq S_B)\), RPC checks the result of the baseline FPU using forward checking, i.e., \(A + B \rightarrow C\) is checked with \(A^H + B^H \rightarrow C^{'},\) and \(A - B \rightarrow C\) is checked with \(A^H - B^H \rightarrow C^{'}\). These two situations can be easily converted to one another to get \(A + (-B) \rightarrow C\). The resulting operation is an addition with same-sign operands \((A\) and \(-B)\). As a result, \(SSADD\) and \(DSSUB\) have identical upper and lower bounds for \(Diff\). Without loss of generality, we explain only \(SSADD\) in this section but the bounds apply equally to \(DSSUB\).

The full-precision FPU computes \(A + B \rightarrow C:\)
\[(S_A \times 2^{E_A} \times M_A) + (S_B \times 2^{E_B} \times M_B) = S_C \times 2^{E_C} \times (M_C + \delta_C)\]

Because \(S_A = S_B = S_C\), we have:
\[2^{E_A} \times M_A + 2^{E_B} \times M_B = 2^{E_C} \times (M_C + \delta_C)\]  
Eqn. (19)

\[M_C' = M_A \times 2^{E_A-E_C} + M_B \times 2^{E_B-E_C} - M_C - \delta_C\]  
Eqn. (20)

The RPC checker computes \(A^H + B^H \rightarrow C^{'}:\)
\[
(S_A \times 2^{E_A} \times M_A^H) + (S_B \times 2^{E_B} \times M_B^H) = S_C' \times 2^{E_C'} \times (M_C' + \delta_C')
\]

Because \(S_A = S_B = S_C'\),
\[2^{E_A} \times M_A^H + 2^{E_B} \times M_B^H = 2^{E_C'} \times (M_C' + \delta_C')\]  
Eqn. (21)

\[M_C' = M_A^H \times 2^{E_A-E_C'} + M_B^H \times 2^{E_B-E_C'} - \delta_C'\]  
Eqn. (22)

As with multiplication, division, and square root, a key aspect of our derivations is to understand the relationship between the exponents of the operands and results. To sum two numbers with the same sign, the mantissa of the operand with the smaller exponent \((E_S)\) must right shift until its exponent matches the operand with the larger exponent \((E_L = \max(E_A, E_B))\). If exactly one bit is to the left of the binary point after summing the two mantissas, then the exponent of the result equals \(E_L\). Else, if more than one bit is to the left of the binary point, then the sum of the mantissas must right-shift accordingly, and the exponent of the result equals \(E_I\) plus the number of right shifts. The sum of the two mantissas reaches its maximum when both operands have the same exponent (i.e., no shift in mantissa of either operand), and this maximum sum of the mantissas is strictly less than 4 (100 in binary) by Axiom 1. As a result, the sum of the two mantissas has at most two bits to the left of the binary point, and thus the number of right shifts is either zero or one.

Based on this analysis, the exponent of the sum, \(E_C\), equals \(E_L\) or \(E_I + 1\). When \(E_C = E_I\), we further know that \(E_A \neq E_B\) because, if \(E_A = E_B\), then we can add \(M_A\) and \(M_B\) without shifting. The smallest value \(M_A + M_B\) can be is 2 (10 in binary) because \(M_A, M_B \geq 1\) according to Axiom 1. Thus, the sum must right shift to normalize \(M_C\), which violates \(E_C = E_I\).

For addition, the exponent of the sum is equal to or one greater than the maximum of the operands’ exponents \((E_i)\). Furthermore, when the exponent of the sum equals \(E_i\), the two operands must have unequal exponents.

Applying this rule to the FPU and checker, we have:

\[ \text{FPU: } \max(E_A, E_B) = E_C \text{ and } E_A \neq E_B \]
\[ \text{ or } \max(E_A, E_B) + 1 = E_C \]

\[ \text{Checker: } \max(E_A, E_B) = E_C' \text{ and } E_A \neq E_B \]
\[ \text{ or } \max(E_A, E_B) + 1 = E_C' \]

The absolute value of \(C'\) must be less than or equal to the absolute value of \(C\). Therefore,
\[ E_C \leq E_C' \]

Similar to multiplication and division, there are 3 types of relationships between \(E_A, E_B\), and \(E_C\).

**Common Case 1:** \(\max(E_A, E_B) = E_C' = E_C\)  
Eqn. (23)

**Common Case 2:** \(\max(E_A, E_B) + 1 = E_C' = E_C\)  
Eqn. (24)

**Corner Case:** \(\max(E_A, E_B) + 1 = E_C' + 1 = E_C\)  
Eqn. (25)

The common cases are considered in Sections 7.1.1 and 7.1.2. For both common cases, \(E_C = E_C'\), so Axiom 6 is valid and the difference between Eqn. (20) and Eqn. (22) can be simplified to:
\[ M_B^H - M_C' = M_A^H \times 2^{E_A-E_C'} + M_B^H \times 2^{E_B-E_C'} - M_C^H - \delta_C' \]
Eqn. (26)

We present the corner case for Eqn. (25) when \(E_C \neq E_C'\) in Appendix B.

7.1.1 Common Case 1

In Eqn. (26), \(A\) and \(B\) are symmetric, and without loss of generality, we can explain only the case when \(\text{abs}(A) \geq \text{abs}(B)\) and \(E_A \geq E_B\) because \(E_A \neq E_B\).

\[ E_C = E_C' = E_A > E_B \]

Eqn. (27)

Eqn. (26) can be simplified to:
\[ M_B^H - M_C' = M_A^H + M_B^H \times 2^{E_B-E_C'} + [M_C^H + \delta_C'] \]

Because \(E_C > E_B\), then \(E_C \geq E_B + 1\):
\[ 0 < 2^{E_B-E_C'} \leq 2^{-1} \]

Using Axiom 3, we can bound <1>:
\[ 0 < M_A^H + M_B^H \times 2^{E_B-E_C'} \leq (2^{-k} - 2^{-23}) + (2^{-k} - 2^{-23}) \times 2^{-1} \]
\[ = 1.5 \times (2^{-k} - 2^{-23}) \]

and we can bound <2>:
\[ -2^{-k} + 2^{-23} \leq -M_B^H \leq 0 \]

Using Axiom 4 and Axiom 5, we can bound <3>:
\[ -2^{-k} - 2^{-24} \leq \delta_C' - \delta_C \leq -2^{-k+1} - 2^{-24} \]

As a result, the upper bound of \(Diff\) is:
\[ (M_B^H - M_C') \times 2^k \leq (1.5 \times (2^{-k} - 2^{-23}) + 2^{-k+1} + 2^{-24}) \times 2^k < 2 \]

The lower bound of \(Diff\) is
\[ (M_B^H - M_C') \times 2^k \geq (-2^{-k} + 2^{-23} - 2^{-k+1} - 2^{-24}) \times 2^k > -1.5 \]

Therefore, the allowable range of \(Diff\) is \([-1.1]\).

7.1.2 Common Case 2

Without loss of generality, we can again explain only \(\text{abs}(A) \geq \text{abs}(B)\) and \(E_A \geq E_B\).

\[ E_C - 1 = E_C' - 1 = E_A \geq E_B \]

Eqn. (28)

Eqn. (26) can be simplified to
\[ M_B^H - M_C' = [M_A^H \times 2^{-1} + M_B^H \times 2^{E_B-E_C'}] + [M_C^H - \delta_C'] \]

Using Axiom 3 and Eqn. (28), we bound <1>:
\[0 < M_A^2 \times 2^{-1} + M_B^2 \times 2^{E_B-E_C} < (2^{-k} - 2^{-23}) \times 2^{-1} \times 2 \]

\[= 2^{-k} - 2^{-23}\]

Terms <2> and <3> have the same boundary conditions as in Section 7.1.1.

So the upper bound of Diff is:

\[(M_B^2 - M_C^2) \times 2^k \leq (2^{-k} - 2^{-23} + 2^{-k-1} + 2^{-24}) \times 2^k < 1.5\]

The lower bound of Diff is still greater than -1.5. Therefore, the allowable range of Diff is [-1,1].

### 7.2 Different-Sign Addition (DSADD) or Same-Sign Subtraction (SSSUB)

Recall that our simple example in Section 3.2 was a SSSUB that motivated reverse checking. The situation occurs when \(A = E_B\) and \(M_B^2 = M_C^2\), but \(M_A^2 \neq M_B^2\). In this situation, the result of \(A - B \to C\) computed by the full-precision FPU has a non-zero value, whereas \(A - B \to C\) as computed by the checker (i.e., with forward checking) equals zero. As a result, Diff = \(C(30;[23-k])C[7+k,0]\) can become arbitrarily large even in the absence of errors. DSADD is equivalent to SSSUB and thus suffers from the same problem in forward checking.

To resolve the challenge, the checker applies reverse checking using \(C^H\) as one of its operands. In this way, we use SSADD/DSSUB to check SSSUB/DSADD.

For SSSUB: \(A - B \to C\), \(S_A = S_B\):

- If \(S_A = S_B = S_C\), check \(C^H + B^H \to A^H\) (SSADD).
- If \(S_A = S_B \neq S_C\), check \(A^H - C^H \to B^H\) (DSSUB).

Similarly, for DSADD \(A + B \to C\), \(S_A \neq S_B\):

- If \(S_A = S_C \neq S_B\), check \(C^H - B^H \to A^H\) (DSSUB).
- If \(S_A \neq S_C = S_B\), check \(A^H - B^H \to C^H\) (DSSUB).

In the following section, we analyze the first scenario above: SSSUB with \(S_A = S_B = S_C\). The other cases can be verified in a similar manner.

The full-precision FPU computes \(A - B \to C\):

\[(S_A \times 2^{E_A} \times M_A) - (S_B \times 2^{E_B} \times M_B) = S_C \times 2^{E_C} + (M_C + \delta_C)\]

Because \(S_A = S_B = S_C\) in this scenario:

\[M_A^2 - 2^{E_A-E_B} \times M_B^2 = 2^{E_C-E_A} \times (M_C + \delta_C)\]

\[M_A^2 = M_B^2 \times 2^{E_B-E_A} + M_C \times 2^{E_C-E_A} - M_A^2 + \delta_C \times 2^{E_C-E_A}\]

The checker performs \(C^H + B^H \to A^H\):

\[(S_C \times 2^{E_C} \times M_C^H) + (S_B \times 2^{E_B} \times M_B^H) = S_A \times 2^{E_A} + (M_A^H + \delta_A)\]

Because \(S_A = S_B = S_C\),

\[2^{E_C-E_A} \times M_B^H + 2^{E_A-E_B} \times M_B^H = M_A^H + \delta_A\]

\[M_A^H = M_B^H \times 2^{E_B-E_A} + M_B^H \times 2^{E_C-E_A} - \delta_A\]

To understand the relationships between \(E_B\) and \(E_C\) in SSSUB (which also apply to DSADD), consider \(A - B = \tilde{C}\), where \(\tilde{C}\) is the true \(C\) with unlimited precision. Then \(B + \tilde{C} = A\).

From our previous analysis in Section 7.1, we have:

\[E_A = \max(E_B,E_C) \quad \text{and} \quad E_B \neq E_C\]  \hspace{1cm} (1)

\[E_A = \max(E_B,E_C) + 1 \quad \text{or} \quad E_B \neq E_C\]  \hspace{1cm} (2)

From Axiom 4,

\[E_C = E_C \text{ if } \tilde{C} \text{ rounds down} \quad \text{or} \quad E_C = E_C + 1 \text{ if } \tilde{C} \text{ rounds up} \]

We now look at the “cross-product” of possible scenarios:

\[(1\&3), (2\&3), (1\&4), (2\&4): \]

\[E_A = \max(E_B,E_C), E_B \neq E_C\]

\[(2\&3) \quad E_A = \max(E_B,E_C) + 1 \]

The other two scenarios are either impossible or subsumed by a previous scenario.

Scenario (1\&4) differs from Scenario (1\&3) only when \(E_B > E_C\), so \(E_A = \max(E_B,E_C) = E_C\) in (1). Then because of (4), \(E_C = E_C - 1\). So \(E_A = E_C - 1 = \max(E_B,E_C)\). Thus, Scenario (1\&4) is impossible because it implies that abs(C) > abs(A), which is impossible after B is subtracted from C, given that \(S_A = S_B\).

Scenario (2\&4) differs from (2\&3) only when \(E_C > E_B\). Then \(E_A = E_C + 1 = (E_C - 1) + 1 = E_C\). Because \(E_C > E_C\), \(E_A = \max(E_B,E_C)\), so \(E_A = \max(E_B,E_C)\), which is the same as scenario (1\&3).

In subtraction, the exponent of the minuend (A) is equal to or one greater than \(E_C\), which is the maximum of the exponents of the subtrahend (B) and difference (C). Furthermore, when the exponent of the minuend equals \(E_C\), the subtrahend and the difference must have unequal exponents.

Applying this rule to the FPU and applying the rule for addition to the checker, we have:

FPU:

\[E_A = \max(E_B,E_C) \quad \text{and} \quad E_B \neq E_C\]

or \(E_A = \max(E_B,E_C) + 1\)

Checker:

\[E_A' = \max(E_B,E_C) \quad \text{and} \quad E_B \neq E_C\]

or \(E_A' = \max(E_B,E_C) + 1\)

Because of truncation in operands B and C, \(E_A' \leq E_A\).

Therefore,

- **Common Case 1**: \(\max(E_B,E_C) = E_A' = E_A, E_B \neq E_C\)  \hspace{1cm} Eqn. (29)
- **Common Case 2**: \(\max(E_B,E_C) + 1 = E_A' = E_A\)  \hspace{1cm} Eqn. (30)
- **Corner Case**: \(\max(E_B,E_C) + 1 = E_A' = E_A\)  \hspace{1cm} Eqn. (31)

We analyze Eqn. (29) and Eqn. (30) in Sections 7.2.1 and 7.2.2. Because \(E_A = E_A'\) in these common cases, Axiom 6 is valid and the difference between \(M_A^H\) and \(M_B^H\) is:

\[M_A^H - M_B^H = M_B^H \times 2^{E_B-E_A} + M_B^H \times 2^{E_C-E_A} + M_B^H \times 2^{E_C-E_A} - \delta_A\]

\[+ \delta_A \times 2^{E_C-E_A}\]  \hspace{1cm} Eqn. (32)

Eqn. (31) can be proved in similar way as the corner case of SSADD (Appendix A).

#### 7.2.1 Common Case 1

In term <1>, because of Eqn. (29), either \(2^{E_C-E_A}\) or \(2^{E_B-E_A}\) must equal 1 because \(\max(E_B,E_C) = E_A\). The other is less than or equal to \(2^{-1}\) because \(E_B \neq E_C\). As a result, <1> is bounded by:

\[0 \leq M_B^H \times 2^{E_B-E_A} + M_B^H \times 2^{E_C-E_A} \leq (2^{-k} - 2^{-23}) \times 2^{-1}\]

\[= 1.5 \times (2^{-k} - 2^{-23})\]

For the same reason, \(2^{E_C-E_A}\) in <3> is smaller than or equal to 1. So we can bound <3>:

\[-2^{-k-1} - 2^{-24} \leq \delta_A \times 2^{E_C-E_A} + \delta_A \leq 2^{-k+1} + 2^{-24}\]

Term <2> is bounded to \([-2^{-k} - 2^{-23}, 0]\) according to Axiom 3. As <1>, <2>, and <3> have the same boundary conditions as <1>, <2>, and <3> in Section 7.1.1, respectively, the bounds on \(\text{Diff}\) are identical to those in Section 7.1.1. Therefore, \(\text{Diff}\) is in the range \([-1,1]\).
7.2.2 Common Case 2

Because of Eqn. (30), \( 0 < 2^{E_A - E_c} \times 2^{E_c - E_A} \leq 2^{-1} \) in \(<1>\). As a result, \(<1>\) is bounded by:

\[
0 \leq M_B^\epsilon \times 2^{E_A - E_c} + M_c^\epsilon \times 2^{E_c - E_A}
\leq (2^{-k} - 2^{-23}) \times 2^{-1} + (2^{-k} - 2^{-23}) \times 2^{-1}
= 2^{-k} - 2^{-23}
\]

In addition to the boundary condition from Axiom 4 and Axiom 5, \(<3>\) is bounded by:

\[
-2^{-k-1} - 2^{-25} < \delta_c \times 2^{E_c - E_a} + \delta_a < -2^{-k-1} + 2^{-25}
\]

Boxed term \(<3>\) has smaller bounds than \(<3>\) in Section 7.1.2. Furthermore, \(<1>\) and \(<2>\) have same upper and lower bounds with \(<1>\) and \(<2>\) in Section 7.1.2, respectively. Therefore, \( Diff \) is in the range \([-1,1]\).

**Summary:** Along with analysis in Appendix A, the analysis in section 7 shows the allowable range of \( Diff \) for addition and subtraction are \([-1,1]\).

Prior work [1] proved a looser bound of \([-2,1]\).

8 Maximum Percentage Error

With bounds on \( Diff \), we can now approximate the Maximum Percentage Error (MPE) for errors undetected by RPC. This derivation is an approximation in that it applies to all common situations but not some of the extremely rare corner cases. This section shows that the undetected errors are small. We compute the percentage error as:

\[
\text{%Error} = \left| \frac{\text{Correct Result} - \text{Erroneous Result}}{\text{Correct Result}} \right| \times 100\%
\]

We have proven that \( Diff \) is in the range of \([-1,1]\) for addition/subtraction and \([-1,3]\) for multiplication, division, and square root. For the common cases, \( Diff \) corresponds to the difference between the fractions of the baseline FPU and the checker.

We represent an erroneous computation with an “E” above the right-arrow and an erroneous result as \( C_e \).

8.1 Forward Checking

In forward checking, the FPU and the checker compute

\[
A \oplus B \rightarrow C_e \text{ and } A^n \oplus B^H \rightarrow C',
\]

\[
\text{MPE} = \left| \frac{C - C_e}{C} \right| = \left| \frac{M_c - M_c^\epsilon}{M_c} \right| \approx \left| \frac{M_c^\epsilon - (M_c^\epsilon)^H}{M_c} \right| \approx \left| \frac{Diff \times 2^{-k}}{M_c} \right|
\]

RPC only misses an error when \( Diff \) is within the derived bounds. Also \( M_c \geq 1 \), so

\[
\text{MPE} \leq \max (|Diff|) \times 2^{-k} \times 100\%
\]

8.2 Reverse Checking

In reverse checking of division (which also applies to square root), \( Diff \) represents the difference in the fractions of \( A^H \) and \( A' \). However, we are really interested in is the difference between \( C_e \) and \( C \). We know that:

\[
A \rightarrow C_e \text{ and } C_e^H \times B^H = A'
\]

The percentage error is:

\[
\left| \frac{C - C_e}{C} \right| \approx \left| \frac{A^H + B^H - A' + B^H}{C} \right| \approx \left| \frac{(A^H - A') + B^H}{C} \right|
= \left| \frac{(Diff \times 2^{-k} + M_b)}{M_c \times 2^{E_c}} \right|
\]

As \( E_A - E_B \geq E_c \) and \( M_b M_c \geq 1 \), MPE is loosely bounded by

\[
\text{MPE} \leq \frac{(Diff \times 2^{-k})}{M_b M_c} \leq \frac{|Diff| \times 2^{-k} \times 100\%}{M_b M_c}
\]

In reverse checking of \( SSSUB \) (also applies to \( DSADD \)),

\[
A - B \rightarrow C_e \text{ and } C_e^H + B^H = A'
\]

Then

\[
\left| \frac{C - C_e}{C} \right| \approx \left| \frac{A - B - (A' - B^H)}{C} \right|
\]

When \( E_A \equiv E_c \equiv E_B \), \( |C - C_e| \equiv |A^H - A'| = |M_a^H - M_a'| \times 2^{E_c}

\[
\text{MPE} = \max \left( \frac{|M_a^H - M_a'|}{M_c} \right) \times 100\% \leq \frac{|Diff|}{M_b M_c} \times 2^{-k} \times 100\%
\]

When \( E_B \equiv E_c \equiv E_A \), \( |C - C_e| \equiv B^L = M_b^L \times 2^{E_c}

\[
\text{MPE} = \max \left( \frac{M_b^L}{M_c} \right) \times 100\% < 2^{-k} \times 100\%
\]

8.3 Summary

Because \( \max (|Diff|) \) for \( SSSUB \) is just 1, MPE is approximately \( \max (|Diff|) \times 2^{-k} \times 100\% \) for all operations. This approximate analysis does not completely bound the maximum percentage error uncaught by RPC, but rather shows that, under most circumstances, undetected errors have very small percentage errors. For example, the MPE for a 16-bit checker \((k=7)\) is 0.78% for addition and subtraction, and 2.34% for the other operations.

9 Implementation and Integration into Processor Core

We implemented RPC as an extension of a floating point unit developed by Kwon et al. [3]. Our implementation of RPC includes the following four components: one \( k \)-bit adder/subtractor, one \( k \)-bit multiplier, logic to determine how to perform the checking, and buffers to hold operands and results until checking can be performed.

9.1 Handling Floating Point Exceptions

When certain floating point exceptions occur, RPC is unable to check for errors. The reason for this limitation is that the assumptions we make about rounding error are valid only when the FPU does not encounter the following exceptions: overflow, underflow, invalid, and divide-by-zero. In these circumstances, the result of the FPU is formatted to positive/negative infinity, zero or denorm, NaN, and positive/negative infinity, accordingly. Therefore, our checker is suppressed when these rare cases are detected.

9.2 Performance Impact

RPC can impact performance if the processor core is waiting for a result to be checked before it can commit a floating point instruction. For some floating point operations, we can perform checking in parallel with the full-precision FPU, but reverse checking does not permit this parallelism. If we try to begin each check as soon as it has its operands, we could encounter a situation in which a reverse check and a forward check need to use the same checker at the same time. To avoid the complexity of managing these situations, we choose to have the checker always wait until the FPU has completed (even for forward checking).
As the checker is modified based on the baseline FPU, it should have at least the same throughput and thus the performance impact of RPC is limited to the extra reorder buffer (ROB) pressure incurred by floating point instructions that are ready to commit but have not yet been checked. The performance penalty due to this extra ROB pressure depends on the microarchitecture and the software running on it, but we do not expect it to be large.

10 \textbf{ERROR DETECTION COVERAGE}

The primary goal of our experimental evaluation is to determine the error detection coverage of RPC.

\subsection*{10.1 Error Injection Methodology}

To evaluate the error detection coverage, we ran an extensive set of error injection experiments. In each experiment, we forced a single stuck-at error on a different wire in the flattened netlist that includes the full-precision FPU and all of the RPC hardware. Note that a single stuck-at error on a wire can often cause multiple errors downstream of the injected error, due to fan-out,
and thus injecting errors at this low level is far more realistic than injecting errors in a microarchitectural simulator [4].

For every wire in the netlist, we ran 2000 experiments. Half of the experiments inject a stuck-at-0 error on the wire and test 1000 different inputs; the other half inject a stuck-at-1 on the wire and test the same 1000 random inputs.

We considered injecting transient errors, but a very large fraction of transient errors were masked (i.e., had no impact on execution). Masking is quite common, as in prior work in error injection [8], because each error affects the results for only a subset of all possible inputs, and often these subsets are tiny. Transient errors, in particular, are masked with very high probability. To obtain statistically significant data in a tractable amount of time, we injected permanent stuck-at errors, which are less likely to be masked. Moreover, a floating point operation is a relatively short latency event, which makes transient errors similar to permanent errors.

10.2 Results

In Figure 3, we show our results by classifying errors into categories, and we present a separate graph for each of the five arithmetic operations (add, sub, mul, div, sqrt). The figures focus on the errors that are unmasked, i.e., those errors that affect the result of the FPU and/or the result of the checker.

Among the unmasked errors, we classify the errors into four categories:

- unmasked and detected (UMD) – our desired outcome
- unmasked undetected (UMUD) – a silent data corruption, which is the worst outcome
- unmasked unchecked (UMUC) – unusual corner case when RPC suppresses checking
- false positive (FP) – error affected checker and is detected even though result of FPU is correct

The first observation from Figure 3 is that there is a clear trade-off between error detection capability and cost. As the checker is made wider, there are fewer undetected errors.

A corollary to this first result is that there are also more false positives, because the checker is larger and thus more liable to be the victim of an injected error. Notice that there are fewer false positives in division and square root than addition, subtraction, and multiplication; this is because the divider and square root unit are significantly larger than the checker multiplier. Unlike a dual modular redundancy scheme (i.e., simply replicate the unit to be checked), which has 50% false positives, RPC can reduce the fraction of false positives by having a smaller checker.

From Figure 3, it appears RPC misses a large portion of unmasked errors at small checker width (such as 16, i.e., 7 bits mantissa). However, Figure 4 shows that RPC detects the vast majority of the large errors.

Figure 4 presents the distribution of the size of the unmasked undetected (UMUD) errors. In these graphs, the top, middle, and bottom bars of the vertical box indicate the first quartile, median, and second quartile of percentage errors, respectively. The top whisker defines a 1.5 inter-quartile range away from the first quartile, and data above here are defined as outliers. The connected green diamonds mark the approximate MPE computed in Section 8. The percentage number above the diamonds is the percentage of UMUD errors that has percentage error larger than the approximate MPE. An error larger than the approximate MPE includes the corner case scenarios and faults in a handful of components that, when faulty, can cause extremely strange behavior. For example, an error in the wire that determines whether the output should be formatted in an exceptional way (e.g., as a NaN) can cause the output to differ dramatically from the correct result and cause an outlier in the percentage error calculation.

We observe that only a very small fraction of UMUDs have percentage errors that are not bounded by the MPE. Overall, the percentage errors of UMUDs are extremely small; the median is less than 0.1% even with a minimally-sized 1-bit mantissa (10-bit checker) for all operations. At checker width 16, only 0.003%–0.055% undetected errors have percentage error larger than the approximate MPE (0.78% for addition and subtraction, and 2.34% for the other operations). This means a vast majority of those undetected errors in Figure 3 at width 16 are very small.

11. Area, Power, and Energy Overheads

We now evaluate the area and power overheads of our RPC implementation.

11.1 Area

We used Synopsys CAD tools to layout the FPU, with and without RPC, in 45nm technology [7]. The results in Figure 5 (a) show that RPC’s area overhead ranges from about 30% to about 90%, depending on the checker’s width. This overhead is far less than that of simple dual modular redundancy (100%).

11.2 Energy

We compute the per operation energy overhead using the power and latency results from the CAD tools. As with the area analysis, we determine the dynamic and static power overheads of RPC with post-synthesis gate-level power estimation. After laying out the circuitry, we obtained the parasitic resistances and capacitances and back-annotated the circuits with them.

To determine the power, we feed the synthesized module with 1000 random inputs for each operation at all checker widths to acquire switching activities of each wire/cell, which is further used to compute both dynamic and static power. To minimize power consumed by buffers for reverse checking, we use clock gating to optimize our design.

Our experiments show that static power comprises roughly 20% of overall power and, unlike dynamic power, is relatively stable across different operations and checker widths.
5(b) shows the per operation energy overhead of 5 operations. Addition and subtraction suffer the most from energy overhead because they are relatively cheap operations. Hence, the checking logic incurs non-trivial overhead. However, before width 18, the overhead is still cheaper than dual modular redundancy. Multiplication incurs somewhat less energy overhead, reaching 100% at width 28. For the most expensive operations, division and square root, as their checker multiplier is significantly smaller in size, RPC is very efficient and has less than 70% overhead at full checker width. Note that the overheads in Figure 5(b) include the impact of the checker, logic, and buffer over each single operation. However, the checker adder is shared between addition and subtraction; the checker multiplier is shared among the other 3 operations; and logic and buffers are shared among all operations. So the overheads do not sum up for a mixture of 5 operations.

12 RELATED WORK

The most related work is the prior work on RPC, which we have already discussed [1][9]. In addition, there have been other proposed schemes for detecting errors in floating point hardware.

Lipetz and Schwarz [5] propose residue checking, which is complete and cost-efficient, in principle, but we cannot resolve how their scheme handles the issue of rounding. We speculate that, based on an IBM patent [2], the FPU passes rounding information to the residue checker, but such a design would be unable to detect errors in rounding logic.

Maniakatos et al. [6] propose a low-cost scheme in which the hardware checks only the exponents of floating point operations. Like us, they seek to detect all large errors, but checking only exponents misses many errors that we consider too large to be acceptable.

13 CONCLUSIONS

In this work, we have applied RPC to an entire floating point unit. We have comprehensively analyzed its ability to detect errors and the size of errors that can go undetected. Based on our results, we believe that RPC is an effective approach for detecting errors in FPUs. We are unaware of other approaches for detecting errors in FPUs that can be tuned to trade off error detection coverage versus cost.

REFERENCES

[1] P. J. Eibl, A. D. Cook, and D. J. Sorin, “Reduced Precision Checking for a Floating Point Adder,” in Proceedings of the 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2009.

[2] S. Iacobovici, “United States Patent: 7769795 - End-to-end residue-based protection of an execution pipeline that supports floating point operations,” U.S. Patent 776979503-Aug-2010.

[3] J.-J. Kwon, J.-S. Moon, J. Sondeen, and J. Draper, “A 0.18 um Implementation of a Floating-point Unit for a Processing-in-memory System,” in Proceedings of the International Symposium on Circuits and Systems, 2004, vol. 2, pp. II-453–6 Vol.2.

[4] M.-L. Li, P. Ramachandran, R. U. Karpuzcu, S. K. S. Hari, and S. Adve, “Accurate Microarchitecture-level Fault Modeling for Studying Hardware Faults,” in Proc. Fourteenth International Symposium on High-Performance Computer Architecture, 2009.

[5] D. Lipetz and E. Schwarz, “Self Checking in Current Floating-Point Units,” in 20th IEEE Symposium on Computer Arithmetic, 2011, pp. 73–76.

[6] M. Maniakatos, Y. Makris, P. Kudva, and B. Fleischer, “Exponent Monitoring for Low-Cost Concurrent Error Detection in FPU Control Logic,” in IEEE VLSI Test Symposium, 2011.

[7] Nangate Development Team, “Nangate 45nm Open Cell Library,” 2012.

[8] A. Pellegrini, R. Smolinski, L. Chen, X. Fu, S. K. S. Hari, J. Jiang, S. V. Adve, T. Austin, and V. Bertacco, “CrashTesting SWAT: Accurate, Gate-Level Evaluation of Symptom-Based Resiliency Solutions,” in Design, Automation Test in Europe Conference Exhibition (DATE), 2012, pp. 1106–1109.

[9] K. Seetharam, L. Co Ting Keh, R. Nathan, and D. J. Sorin, “Applying Reduced Precision Arithmetic to Detect Errors in Floating Point Multiplication,” in Proceedings of the Pacific Rim International Symposium on Dependable Computing, 2013.

Acknowledgments

This material is based on work supported by the National Science Foundation under grant CCF-111-5367.

Appendix A: $E_{C} = E_{C}' + 1$ for SSADD

In this section we consider the corner case scenario for SSADD Eqn. (25). When $E_{C} = E_{C}' + 1$, Axiom 6 is no longer valid. To derive the upper bound of Diff in this case, we need to understand the difference (we denote as D) between what is computed by the FPU and the checker and under what circumstance $D$ causes $E_{C} \neq E_{C}'$. To simplify the analysis, we introduce the notation $|X|$ to denote the integer represented by the concatenation of the exponent and mantissa bits of X without sign bits. So for example, $|C| [30 : (23 - k)]$ and $|C' | [(7 + k):0]$ is $|C'|$.

We know that $|C'| \leq |C|$ because of truncation in operands in computing C. As we care only about the magnitude of the difference in this case, we define $D = |C'| - |C|$. Then $|C'| = |C| + D$. (Note that D is not Diff, which is $|C'| - |C|$. D is the difference between the left-hand sides of Eqn. (19) and Eqn. (21):

$$D = M_{b} \times 2^{E_{b}} + M_{d} \times 2^{E_{d}}$$

Let $D = 2^{E_{b}} \times M_{b}$ and let $E_{b}' = E_{b}'$. Then $M_{b}' = 2^{E_{b}} \times M_{b} + 2^{E_{d} - E_{b}'}$. Hence $E_{b}'$ differs from $E_{b}$ only when $M_{b}' + M_{d} \geq 2$. Then to normalize the sum, $M_{b}' + M_{d}$ must right shift, resulting in $E_{b}' = E_{b}' + 1$. From Eqn. (25), $E_{b}' = max(E_{b}, E_{b}')$. Therefore, $2^{E_{b} - E_{b}'} \leq 1$ and $2^{E_{b} - E_{b}'} \leq 1$, but they cannot both equal 1 at the same time. Thus, we have:

$$M_{b}' = M_{b}' \times 2^{E_{b}} + M_{d} \times 2^{E_{d}}$$

Therefore, $M_{d}$ must be in the form of 0, 0000 - 0001{ x }, where each x represents an unknown bit and can be either zero or one. Each x is independent from each other and they may or may not hold the same value. $M_{d}'$ is all zero after its first k bits to the right of its binary point, i.e., it is in the form 1{ x }_{k}000...
If the sum of the two ever reaches 2, the first x in the fraction of $M_p$ must equal 1 and all x in $M'_c$ must equal 1. Their sum must be in the form of $10.\{k\}$ before being normalized. After being normalized, $M'_c$ must be in the form of $\{k\}$.

$$M_B: \begin{array}{l}
0: 0.0000001 \\
+ M'_c: 1.1111111 0\ldots 0 \\
M_B + M'_c: 10.0000000 x\ldots x \\
M_c: 1.00000000 x\ldots x \\
\end{array}$$

$k$ bits

So when $E_C \neq E'_C$, the fractions (not including the implicit 1) must be $f_i = \{k\}$, now consider $Diff = |C'| - |C|:

$$\begin{array}{l}
|C'| = \{\exists_{\ell} \{f_{\ell}'\}; \exists_{\ell} + 1\{00\ldots 0\} \\
- |C| \rightarrow \{\exists_{\ell} \{f_{\ell}\} \rightarrow \{\exists_{\ell} \{11\ldots 1\} \\
Diff (E_C - E'_C) \times 2^k + f_i' - f_i \times 1 \\
\end{array}$$

where $\exists_{\ell} = E_C + 127$ and $\exists'_\ell = E'_C + 127$. Therefore, for the corner case Eqn. (25), $Diff$ equals 1, which is within the range $[-1, 1]$.

**Appendix B: $E_C = E'_C + 1$ for Multiplication**

This corner case of multiplication, Eqn. (8), follows a similar analysis as in Appendix A. In this case, $D$ is the difference between the left hand sides of Eqn. (2) and Eqn. (4).

$$D = M^* \times 2^k \times 2^e \times b$$

where $M^*$ is defined in Eqn. (9). From Eqn. (8), we know that $E_A + E_B = E'_C$. Let $D = 2^{k_B} + E_B = E'_C$. Then we have $M_B = M^*$. In Section 5.1, we bound $M^*$ in $<1$. So $M_B < 4 \times 2^{-k}$ and is thus in the form of $0.\{1\}_k(-2^\ldots)$). We also know that $M'_c$ has only k mantissa bits, so it is of the form $1.\{x\}_k(0\ldots)$. $M_B$: 0. 000000 xx x x 0 0

$k$ bits

The first $k$ values of x in the fraction of $M'_c$ must be 1 for the sum to reach 2. The sum of the last two x bits of $M_B$ (denoted as $ab$) and last two y bits of $M'_c$ (denoted as $cd$) must be greater than or equal to 4 to produce the carry one.

$$M_B: \begin{array}{l}
0: 0.0000000ab x\ldots x \\
+ M'_c: 1.111111cd 0\ldots 0 \\
M_B + M'_c: 10.000000ef x\ldots x \\
M_c: 1.00000000 ef x\ldots x \\
\end{array}$$

$k$ bits

Now $Diff$ can be evaluated in the following steps:

$$\begin{array}{l}
|C'| = \{\exists_{\ell} \{00\ldots 0\} \rightarrow \{\exists_{\ell} + 1\{00\ldots 0\} \\
- |C| \rightarrow \{\exists_{\ell} \{11\ldots 1\} \rightarrow \{\exists_{\ell} \{11\ldots 1\} \\
Diff 2^k (1_{k+2}0\ldots 0 + 0e - cd) \rightarrow \{1\} \{0e - cd\} = 10e - cd \\
\end{array}$$

Consider all possible value of $ab$, $cd$, and $ef$, where $ab + cd = 1ef$. The two-bit values $ab$ and $cd$ are both less than or equal to $11_2 = 3_{10}$. Therefore, $ef$ cannot be $11_2$, as that requires $ab + cd = 111_2 = 7_{10}$. Thus, the possible values of $ab$, $cd$, $ef$, and $Diff$ are as follows:

| $ab$ | $cd$ | $1ef$ | $10e$ | $10e - cd$ |
|-----|-----|-----|-----|-----|
| 01  | 11  | 100 | 100 | 12 = 1_{10} |
| 11  | 11  | 100 | 100 | 11_2 = 3_{10} |
| 10  | 10  | 100 | 100 | 10_2 = 2_{10} |
| 10  | 11  | 100 | 100 | 12 = 1_{10} |

Therefore, $Diff$ is still within the range $[-1, 1]$ of multiplication.

**Appendix C: Upper Bound of Diff in Multiplication**

In Section 5.1 we proved the upper bound of $Diff$ is less than 4.5, indicating that this integer difference can be as large as 4. However, after billions of simulations, we never observed $Diff$ equaling 4. In this section, we introduce a more sophisticated proof that bounds $Diff$ to be less than 4 (i.e. $M^u_i - M^p_i < 4 \times 2^{-k}$) and thus this integer difference can be at most 3. This proof also implies that the upper bounds on $Diff$ for division and square root are also less than 4.

To bound $Diff$, we must understand how $M^p_i$ differs from $M^u_i$. Thus, we need to understand how rounding happens in the checker and how rounding in the FPU affects the most significant $k$ bits in $M_c$ (i.e. $M^u_i$).

In the following analysis, we denote the unrounded (unlimited precision) result of the FPU’s mantissa as $M_p$, and the unrounded result of the checker’s mantissa as $M'_p$. With unlimited precision, $M_p$ has more than 23 bits of fraction (i.e., bits after the binary point), and $M'_p$ has more than $k$ bits of fraction even though their input operands have only 23 and $k$ bits of fractions, respectively. We split both $M_p$ and $M'_p$ into High and Low parts: $M_p = M^u_i + M^l_i$ and $M'_p = (M^u_i) + (M^l_i)$, where $M^u_i$ refers to the nearest integer and the first $k$ bits of the fraction; $M^l_i$ refers to the bits less significant than $M^u_i$. For example, $M = 1.\{x\}_k$, $M^u_i = 1.\{x\}_k$, $M^l_i = 0.\{0\}_k(\ldots)$.

We derive an upper bound on $M^u_i - M^l_i$ in three steps:

**Step 1:** Analyze the relationship between $M^u_i$ and $(M^l_i)^+$. $M^u_i = M^l_i + M^*$. $M^l_i$ is defined to be $M^u_i + M^l_i$ and is the range $[0, 2 \times 2^{-k})$. We sub-divide this range into two sub-ranges for purposes of our analysis:

- **Range 1:** $0 \leq (M^u_i)^+ + (M^l_i)^+ < 2^{-k}$
- **Range 2:** $2^{-k} \leq (M^u_i)^+ + (M^l_i)^+ < 2 \times 2^{-k}$

In Range 1, the sum of $(M^u_i)^+ + (M^l_i)^+$ does not cause a carry into the sum of the High terms $(M^u_i)^+ + (M^l_i)^+$ in Eqn. (33). So:
We now have to consider all possible combinations of cases from Steps 1, 2.a, and 2.b. Because there are two cases in each step, there are $2^3=8$ theoretical combinations, but some combinations are impossible.

- **Case I-Case III-Case V:** Plug in the equations for the high bits in Case III and Case V into Case I.
  \[
  M_H^I = M_c' + (M^*)^H 
  \]
  Rearrange the equation to get:
  \[
  M_c^I - M_c = (M^*)^H < 4 \times 2^{-k} 
  \]

- **Case I-Case IV-Case V (impossible):** $M_c^I = (M_c')^I + (M^*)^I$ from Case I. $M_c^I$ is in the range $(2^{-k} - 2^{-24}, 2^{-k})$, and $(M_c')^I$ is in the range $[0, 2^{-k-1})$. Then the boundary condition for $(M^*)^I$ is
  \[
  2^{-k} - 2^{-24} < (M^*)^I < 2^{-k-1} 
  \]
  Since the upper bound is smaller than the lower bound, because $k<24$, there is no $(M^*)^I$ that satisfies the requirements of Case I-Case IV-Case V, i.e. the case does not exist.

- **Case I-Case III-Case VI:** Case III and Case VI into Case I.
  \[
  M_H^I = M_c' + (M^*)^H - 2^{-k} 
  \]
  \[
  M_c^I - M_c = (M^*)^H - 2^{-k} < 3 \times 2^{-k} 
  \]

- **Case I-Case IV-Case VI:** Plug in Case IV and Case VI into Case I:
  \[
  M_c^I - 2^{-k} = M_c' + (M^*)^H - 2^{-k} 
  \]
  \[
  M_c^I - M_c = (M^*)^H < 4 \times 2^{-k} 
  \]

- **Case II-Case III-Case V (impossible):** From Case II, $(M_c')^I + (M^*)^I = M_c' + 2^{-k}$, $(M_c')^I + (M^*)^I$ is in the range $(2^{-k} - 2^{-24}, 2^{-k})$, and $(M_c')^I$ is in the range $[0, 2^{-k-1})$. Then the boundary condition of $(M^*)^I$ is
  \[
  2^{-k} - 2^{-24} < (M^*)^I < 2^{-k-1} 
  \]
  which is invalid because $(M^*)^I < 2^{-k}$. Therefore, Case II-Case III-Case V is impossible.

- **Case II-Case III-Case VI:** Plug Case III and Case VI into Case II:
  \[
  M_c^I - 2^{-k} = M_c' + (M^*)^H - 2^{-k} 
  \]
  \[
  M_c^I - M_c = (M^*)^H < 4 \times 2^{-k} 
  \]

- **Case II-Case III-Case V (impossible):** $(M_c')^I + (M^*)^I = M_c' + 2^{-k}$ from Case II. $(M_c')^I + (M^*)^I$ is in the range $(2 \times 2^{-k} - 2^{-24}, 2 \times 2^{-k})$. $(M_c')^I$ is in the range $[0, 2^{-k-1})$. Then
  \[
  2 \times 2^{-k} - 2^{-24} < (M^*)^I < 1.5 \times 2^{-k} 
  \]
  Because the upper bound is smaller than the lower bound, this case is impossible.

- **Case II-Case IV-Case VI (impossible):** As in Case II-Case III-Case V, $(M^*)^I$ is in the range $(2 \times 2^{-k} - 2^{-24}, 2 \times 2^{-k})$. $(M_c')^I$ is in the range $[2^{-k-1}, 2^{-k})$. Thus
  \[
  1.5 \times 2^{-k} - 2^{-24} < (M^*)^I < 1.5 \times 2^{-k} 
  \]
  Because $(M^*)^I < 2^{-k}$, this case is impossible.

**Step 2.a**

For the FPU, $M_c = M_c + \delta_c$. So
\[
M_c = M_c^I + \delta_c + M_c^I + \delta_c = M_c^I + \delta_c 
\]

The only way in which $M_c^I \neq M_c^I$ is if, after rounding, the carried one at the 23rd bit propagates to the most significant $k$ bits. For this to happen, the $(k+1)^{th}$ to the 23rd bits after the binary point in $M_c$ must all equal 1. To produce the carried one, the bits after the 23rd bit of $M_c$ must be larger than $2^{-24}$, so $M_c$ rounds up to get $M_c$. Thus, $M_c^I \neq M_c^I$ when $M_c^I > 0.0_{10} \{1\}_{23-k}(1) = 2^{-k} - 2^{-24}$. After rounding, all bits in $M_c^I$ must be zero because they all sum with the carried one. For purposes of analysis, we consider two ranges of values for $M_c^I$:

- **Range 1:**
  \[
  0 < M_c^I < 2^{-k} - 2^{-24} 
  \]
  Thus, $M_c^I = M_c^I$ and $M_c^I = M_c^I + \delta_c$

- **Range 2:**
  \[
  2^{-k} - 2^{-24} < M_c^I < 2^{-k} 
  \]
  In Range 2, $M_c$ rounds up to $M_c^I$ and is $2^{-k}$ larger than $M_c^I$ after rounding:
  \[
  \begin{align*}
  M_c^I &= M_c^I - 2^{-k} \\
  M_c^I &= M_c^I + \delta_c + 2^{-k} 
  \end{align*} 
  \]

**Step 2.b**

For the checker,
\[
M_c = (M_c')^I + (M_c')^I = M_c' + \delta_c 
\]

Notice there is no $(M_c')^I$ term, because $M_c'$ is produced by the checker and has only $k$ bits after the binary point. For analysis, we consider two ranges of values for $(M_c')^I$:

- **Range 1:**
  \[
  0 \leq (M_c')^I < 2^{-k-1} 
  \]
  Thus, $(M_c')^I = M_c'$ and $(M_c')^I = \delta_c$

- **Range 2:**
  \[
  2^{-k-1} \leq (M_c')^I < 2^{-k} 
  \]
  In Range 1, $M_c'$ rounds down to get $M_c^I$. $M_c'$ equals $(M_c')^I$ after rounding:
  \[
  \begin{align*}
  (M_c')^I &= M_c' \\
  (M_c')^I &= \delta_c 
  \end{align*} 
  \]

- **Case V:**
  \[
  \begin{align*}
  ((M_c')^I &= M_c' - 2^{-k} \\
  (M_c')^I &= \delta_c + 2^{-k} 
  \end{align*} 
  \]

- **Case VI:**
  \[
  \begin{align*}
  (M_c')^I &= M_c' - 2^{-k} \\
  \delta_c + 2^{-k} 
  \end{align*} 
  \]
Conclusion Overall, $M_C^H - M_C^L < 4 \times 2^{-k}$. Therefore $Diff = (M_C^H - M_C^L) \times 2^{-k}$ in Section 5.1 is in [-1,3].