SOFT ERROR PROPAGATION IN FLOATING-POINT PROGRAMS

by

Sha Li

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Summer 2010

SOFT ERROR PROPAGATION IN FLOATING-POINT PROGRAMS

by

Sha Li

Approved: _____
Xiaoming Li, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____
Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____
Michael J. Chajes, Ph.D.
Dean of the College of Engineering

Approved: _____
Debra Hess Norris, M.S.
Vice Provost for Graduate and Professional Education

ACKNOWLEDGMENTS

During the past two years at the University of Delaware, I have learned a lot. Here, I want to give my thanks to all persons who have helped me.

I am especially grateful to my advisor, Dr. Xiaoming Li, for his guidance and support all the time. He has taught me how to do research and given me directions when I met problems in research. He led me to the subject of this thesis. His wisdom, insightful advice, and the freedom he gave had a significant influence in my work.

I wish to thank all persons in Lab 323 (especially Liang and Murat). We are in the same Lab and we try to help each other all the time.

Most of all, I want to thank my parents, for their love. They always trust me and give me enough freedom to decide what I want to do.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

As technology scales, VLSI performance has experienced an exponential growth. As feature sizes shrink, however, we will face new challenges such as soft errors (single-event upsets) to maintain the reliability of circuits. Recent studies have tried to address soft errors with error detection and correction techniques such as error-correcting codes or redundant execution. However, these techniques come at a cost of additional storage or lower performance.

We present a different approach to address soft errors. We start from building a quantitative understanding of the error propagation in software and propose a systematic evaluation of the impact of bit flip caused by soft errors on floating-point operations. Furthermore, we introduce a novel model to deal with soft errors. More specifically, we assume soft errors have occurred in memory and try to know how the errors will manifest in the results of programs. Therefore, some soft errors can be tolerated if the error in result is smaller than the intrinsic inaccuracy of floating-point representations or within a predefined range. We focus on analyzing error propagation for floating-point arithmetic operations.

Our approach is motivated by interval analysis. We model the rounding effect of floating-point numbers, which enable us to simulate and predict the soft error propagation for a single floating-point arithmetic operation. In other words, we model and simulate the relation between the bit flip rate, which is determined by soft errors in hardware, and the error of floating-point arithmetic operations. And the simulation results enable us to tolerate certain types of soft errors without expensive error detection and correction processing.

Chapter 1

INTRODUCTION

Soft errors are unexpected changes of the states in a computer system. Usually they are one-time events and can occur in both memory and logic circuits. Soft errors are traditionally caused by natural phenomena such as radiation. Therefore, the study of soft errors is limited to applications in extreme environments such as space or high-altitude aviation. As the degree of integration increases rapidly and the power density of circuits increases even faster, soft errors are increasingly triggered by some inherent properties of the circuit such as high temperatures during the execution of a computational intensive program, susceptibility of the circuit to voltage fluctuations, or the inherent variation of transistor physics introduced in the manufacturing process [1].

The problem caused by soft errors can be attacked from hardware, architecture or software. The basic idea is to detect a problem-causing soft error and to recover from the fault. Error correcting codes (ECC) is probably the most widely employed technology to detect the unexpected error happening in memory [2] or in microprocessors [3, 4]. It can be designed to recover from 1-bit error, though at the cost of increased storage and longer latency. Soft errors can also be detected or recovered by executing a program redundantly either in time or space. The redundant execution in time means to execute the same program at different times. The spatially redundant execution can be achieved by running the program on different CPUs or compiling the program into semantically equivalent binary codes that have different instruction compositions or scheduling. Redundant execution can work because soft errors are transient, i.e., the same soft error will not repeat in a

short period of time on the same circuit. To detect a soft error, a program is usually executed twice [5] and the results are compared. To further recover from a soft error, some kind of voting is necessary to choose the likely correct result from more than two redundant executions [6, 7]. The efficiency of redundant execution can be improved by either repeating only the critical part of a program [8] or exploiting the intrinsic semantic of the program to avoid the redundant execution of the whole program [9, 10].

No matter what error detection or error recovery techniques are used, the ultimate goal is to prevent a program from producing wrong results. However, the requirement of absolute correctness comes with a price and is not always necessary. Most techniques use more storage. For example, ECC needs extra bits to store the correction code. The redundant execution technique can double the memory footprint of that of a normal run. Furthermore, error detection and recovery need additional processing of data, and therefore introduce overhead. For example, the ECC memory is generally slower than the memory with the same manufacturing technology. Redundant execution can slowdown a program to less than 50% of its original speed. Overall, those are the price for the assurance of absolutely removing the effect of soft error from the result of programs.

We propose a novel approach to evaluate the impact of soft error on floating-point operations. Our study is motivated by the question "Is it possible to tolerate certain kinds of soft error?" In other words, under some conditions, the final output of a program will not be affected by the faults triggered by certain kinds of soft error. If so, what kinds of soft error can be tolerated? We focus our study here on floating-point operations because the digital representation of floating-point numbers is discrete and has limited range. Therefore, from a high-level point of view, the floating-point operations in a computer come with intrinsic inaccuracy. If the deviation of value caused by a soft error is smaller than the intrinsic inaccuracy of a floating-point operation, such soft errors can be tolerated by that floating-point operation. Therefore, as a consequence of the intrinsic inaccuracy, the result of floating-point program is usually considered acceptable as long as

it falls into a pre-specified range. That further relaxes the tolerance of soft errors. Overall, the novelty of our approach and also our main contribution is to identify soft errors that can be tolerated considering the intrinsic inaccuracy and the semantically acceptable error range in floating-point operations. More specifically, we model the error propagation of soft error in the four basic arithmetic floating-point operations, and use simulation to build a quantitative and empirical predictor that can predict what kinds of soft error can be tolerated.

1.1 Outline

This thesis is organized as follows:

In Chapter 2, we describe our approach to evaluate the impact of soft error on floating-point operations. We first give the general modeling method we use to analyze soft error propagation. Based on the general modeling method, we analyze bit flip error propagation in addition, subtraction, multiplication, and division. For each of these four operations, we first show the analysis results, then give algorithms to compute the error propagation results.

In Chapter 3, we show the bit flip error propagation results for four basic floating-point operations. Besides giving the results for four basic floating-point operations, we also present the relation between precision and threshold.

Finally, we conclude the thesis and give some future directions to continue the current work.

Chapter 2

ANALYSIS

We approach the tolerance of soft errors in two steps. The first step is to model the propagation of the bit flip error in basic floating-point arithmetic operations. More specifically, the problem can be formalized as the following:

- For two numbers A and B, define $C = AopB$ (op represents one of the four basic arithmetic operations). If soft error causes bit flip in operand A or B (A becomes A', or B becomes B'), then the result C becomes C' ($= A'opB$, or $= AopB'$).
- Question: what is the distribution of $(C' - C)$ after single arithmetic operation?

In addition, we assume that both two operands and the result are floating-point numbers that conform to standard floating-point formats.

The second step is the simulation of error propagation by applying the model to all categories that are determined from the model. We first describe the general idea of our approach in Section 2.1. After that, the modeling and the simulation of error propagation for the four basic floating-point arithmetic operations will be discussed in detail from Section 2.2 to Section 2.5.

2.1 General Modeling Method

Our analysis is motivated by the interval analysis method [11, 12]. We first need to define the representation of floating-point formats in the model. Floating-point formats are characterized by their radix, precision, and exponent range. We need to define these three parameters. For radix, because the error we consider is caused by bit flip, we use radix *two* - zero and one. For precision, we define it as p - the number of bits in the significand. For exponent range, we use two parameters, *emax* - the maximum exponent, and *emin* - the minimum exponent. Moreover, according to the IEEE754 standard, which is the most widely used standard for floating-point computation, *emin* shall be $(1 - emax)$ for all formats.

Due to the discrete nature of the floating-point representation in computers, floating-point arithmetic is only a systematic approximation of real arithmetic. We first need to model this approximation. Floating-point arithmetic can only represent a finite subset of the continuum of real numbers, so after one operation, we have to modify the result to fit it in the format while signaling the inexact exception, underflow, or overflow when necessary. Moreover, according to IEEE754, every operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result. Thus, we need take into consideration two effects in order: normalization and rounding. For normalization, we always try to keep only one non-zero bit left to the binary point by adjusting the exponent value (suppose we only consider non-subnormal floating-point number). We denote the normalized number as *num*. For rounding, suppose we use the default rounding mode - round to nearest, the rounding error is no more than half ulp (which means Unit in Last Place). Therefore, we can use an interval to represent the floating-point number after these two effects - $[num - 0.5ulp, num + 0.5ulp]$. The interval contains the actual floating-point number. It also represents the intrinsic inaccuracy of the floating-point operation in computers. Furthermore, since we use radix two to represent floating-point number, bit flip caused by

soft errors can be modeled as $0 \rightarrow 1$ or $1 \rightarrow 0$.

Next, we will analyze bit flip error propagation for four basic floating-point arithmetic operations — addition, subtraction, multiplication, and division. Moreover, we assume bit flip only happens in fraction part which means no bit flip in exponent part or sign bit. And the analysis is only for a single floating-point arithmetic operation.

2.2 Bit Flip Error Propagation in Addition

To begin with, we define A, B and C as the following (same definitions will be used in the analysis for Subtraction, Multiplication, and Division):

$$A = 1.a_1a_2\cdots a_{p-1} \times 2^{E_A},$$

$$B = 1.b_1b_2\cdots b_{p-1} \times 2^{E_B},$$

$$C = 1.c_1c_2\cdots c_{p-1} \times 2^{E_C}.$$

Here, p is the precision, $\{a_1,\cdots,a_{p-1},b_1,\cdots,b_{p-1},c_1,\cdots,c_{p-1}\} \in \{0,1\}$, and $emin \leq \{E_A,E_B,E_C\} \leq emax$.

Due to commutability of addition, we can start with either A or B. Here we assume it is A that has bit flip. Then, A' can be represented as the following:

$$A = 1.a'_1a'_2\ldots a'_{p-1} \times 2^{E_A}.$$

Here, the exponent is the same with A's (still E_A), and $\{a'_1,\cdots,a'_{p-1}\} \in \{0,1\}$. Because addition is commutative, we only need consider $A'+B$ (or $A+B'$).

Next, let's express C in terms of A and B. The addition of two binary numbers conceptually consists of multiple steps. The first step is to align A and B according to their exponent values. Moreover, after adding A and B, the result may be not in a normalization form which means the leftmost non-zero bit may not be immediately left to the binary point (because of the carry). In that case, we need to shift the result C to normalize it and accordingly increase the exponent value. After normalization, the intermediate result will be rounded, which can be modeled by using an interval to represent the result. Therefore, C may have the expressions as shown in Table 2.1 depending on whether there is normalization or carry.

The difference between C and C' comes from one of the two operands, where we use A' for C' instead of A (same B for both C and C'). The expressions for C' are shown in Table 2.2. Note: In Table 2.1 and Table 2.2, "No carry" means there is NOT any carry to

the non-zero bit left to the binary point; "Yes carry" means there is carry to the non-zero bit left to the binary point.

Table 2.1: Expressions for $C\ (=A+B)$

1. $E_A > E_B$	1.1. No carry $[1.a_1a_2\cdots a_{p-1}+0.\underbrace{00\cdots01}_{d}b_1b_2\cdots b_{p-1}-0.\underbrace{00\cdots00}_{p-1}1,$ $1.a_1a_2\cdots a_{p-1}+0.\underbrace{00\cdots01}_{d}b_1b_2\cdots b_{p-1}+0.\underbrace{00\cdots00}_{p-1}1]\times 2^{E_A}$ 1.2. Yes carry $[0.1a_1a_2\cdots a_{p-1}+0.0\underbrace{00\cdots01}_{d}b_1b_2\cdots b_{p-1}-0.\underbrace{00\cdots00}_{p-1}1,$ $0.1a_1a_2\cdots a_{p-1}+0.0\underbrace{00\cdots01}_{d}b_1b_2\cdots b_{p-1}+0.\underbrace{00\cdots00}_{p-1}1]\times 2^{E_A+1}$ $(d=E_A-E_B)$
2. $E_A = E_B$	$[0.1a_1a_2\cdots a_{p-1}+0.1b_1b_2\cdots b_{p-1}-0.\underbrace{00\cdots00}_{p-1}1,$ $0.1a_1a_2\cdots a_{p-1}+0.1b_1b_2\cdots b_{p-1}+0.\underbrace{00\cdots00}_{p-1}1]\times 2^{E+1}$ $(E=E_A=E_B)$
3. $E_A < E_B$	3.1. No carry $[0.\underbrace{00\cdots01}_{d}a_1a_2\cdots a_{p-1}+1.b_1b_2\cdots b_{p-1}-0.\underbrace{00\cdots00}_{p-1}1,$ $0.\underbrace{00\cdots01}_{d}a_1a_2\cdots a_{p-1}+1.b_1b_2\cdots b_{p-1}+0.\underbrace{00\cdots00}_{p-1}1]\times 2^{E_B}$ 3.2. Yes carry $[0.0\underbrace{00\cdots01}_{d}a_1a_2\cdots a_{p-1}+0.1b_1b_2\cdots b_{p-1}-0.\underbrace{00\cdots00}_{p-1}1,$ $0.0\underbrace{00\cdots01}_{d}a_1a_2\cdots a_{p-1}+0.1b_1b_2\cdots b_{p-1}+0.\underbrace{00\cdots00}_{p-1}1]\times 2^{E_B+1}$ $(d=E_B-E_A)$

From Table 2.1 and Table 2.2, we know the pair of (C', C) may have the following different combinations:

- When $E_A > E_B$, (C', C) may be (1.1, 1.1), (1.2, 1.2), (1.1, 1.2), (1.2, 1.1);
- When $E_A = E_B$, there is only one case which is (2, 2);

Table 2.2: Expressions for C' $(= A' + B)$

1. $E_A > E_B$	1.1. No carry $[1.a'_1a'_2\cdots a'_{p-1} + 0.\underbrace{00\cdots01}_{d}b_1b_2\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $1.a'_1a'_2\cdots a'_{p-1} + 0.\underbrace{00\cdots01}_{d}b_1b_2\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A}$ 1.2. Yes carry $[0.1a'_1a'_2\cdots a'_{p-1} + 0.0\underbrace{00\cdots01}_{d}b_1b_2\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $0.1a'_1a'_2\cdots a'_{p-1} + 0.0\underbrace{00\cdots01}_{d}b_1b_2\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A+1}$ $(d = E_A - E_B)$
2. $E_A = E_B$	$[0.1a'_1a'_2\cdots a'_{p-1} + 0.1b_1b_2\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $0.1a'_1a'_2\cdots a'_{p-1} + 0.1b_1b_2\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E+1}$ $(E = E_A = E_B)$
3. $E_A < E_B$	3.1. No carry $[0.\underbrace{00\cdots01}_{d}a'_1a'_2\cdots a'_{p-1} + 1.b_1b_2\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $0.\underbrace{00\cdots01}_{d}a'_1a'_2\cdots a'_{p-1} + 1.b_1b_2\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_B}$ 3.2. Yes carry $[0.0\underbrace{00\cdots01}_{d}a'_1a'_2\cdots a'_{p-1} + 0.1b_1b_2\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $0.0\underbrace{00\cdots01}_{d}a'_1a'_2\cdots a'_{p-1} + 0.1b_1b_2\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_B+1}$

- When $E_A < E_B$, (C', C) may be (3.1, 3.1), (3.2, 3.2), (3.1, 3.2), (3.2, 3.1).

Here, the number 1.1 to 3.2 are the case index as shown in tables 2.1 and 2.2.

After getting the expressions for C and C', we can represent the error in the result. The error is the difference between C and C', i.e. $\Delta_C = C' - C$. Although we have assumed that no bit flip occurs in exponent part of operands, C and C' may still have different exponent value , because C may have carry, while C' may not, and vice versa. Hence, if C and C' have different exponent values, we need align them first (shift the one having smaller exponent value), then subtract them. Otherwise, we directly subtract them. The

error in the result is shown in Table 2.3. Note: 1. for (1.*, 1.*), d is defined as $d = E_A - E_B$; 2. for (3.*, 3.*), d is defined as $d = E_B - E_A$; 3. Δ_A is defined as $\Delta_A = 1.a'_1a'_2\cdots a'_{p-1} - 1.a_1a_2\cdots a_{p-1}$.

Table 2.3: Error of Addition Operation ($C' - C$)

(1.1, 1.1)	$[\Delta_A - 0.\underbrace{00\cdots00}_{p-2}1, \Delta_A + 0.\underbrace{00\cdots00}_{p-2}1] \times 2^{E_A}$
(1.2, 1.2)	$[\Delta_A \times 2^{-1} - 0.\underbrace{00\cdots00}_{p-2}1, \Delta_A \times 2^{-1} + 0.\underbrace{00\cdots00}_{p-2}1] \times 2^{E_A+1}$
(1.1, 1.2)	$[\Delta_A \times 2^{-1} - 0.\underbrace{00\cdots00}_{p-1}11, \Delta_A \times 2^{-1} + 0.\underbrace{00\cdots00}_{p-1}11] \times 2^{E_A+1}$
(1.2, 1.1)	$[\Delta_A \times 2^{-1} - 0.\underbrace{00\cdots00}_{p-1}11, \Delta_A \times 2^{-1} + 0.\underbrace{00\cdots00}_{p-1}11] \times 2^{E_A+1}$
(2, 2)	$[\Delta_A \times 2^{-1} - 0.\underbrace{00\cdots00}_{p-2}1, \Delta_A \times 2^{-1} + 0.\underbrace{00\cdots00}_{p-2}1] \times 2^{E+1}$
(3.1, 3.1)	$[\Delta_A \times 2^{-d} - 0.\underbrace{00\cdots00}_{p-2}1, \Delta_A \times 2^{-d} + 0.\underbrace{00\cdots00}_{p-2}1] \times 2^{E_B+1}$
(3.2, 3.2)	$[\Delta_A \times 2^{-(d+1)} - 0.\underbrace{00\cdots00}_{p-2}1, \Delta_A \times 2^{-(d+1)} + 0.\underbrace{00\cdots00}_{p-2}1] \times 2^{E_B+1}$
(3.1, 3.2)	$[\Delta_A \times 2^{-(d+1)} - 0.\underbrace{00\cdots00}_{p-1}11, \Delta_A \times 2^{-(d+1)} + 0.\underbrace{00\cdots00}_{p-1}11] \times 2^{E_B+1}$
(3.2, 3.1)	$[\Delta_A \times 2^{-(d+1)} - 0.\underbrace{00\cdots00}_{p-1}11, \Delta_A \times 2^{-(d+1)} + 0.\underbrace{00\cdots00}_{p-1}11] \times 2^{E_B+1}$

Table 2.3 shows that the error in the result can be purely represented as a function of the error in operand A. At this point, the question is how to model the error distribution of one operand (i.e., the distribution of $(A' - A)$ after bits get flipped).

The origin of the error comes from bit flip. Assume the probability of single bit flip is known. Since bit flip occurs in operand A, according to single bit flip error probability we can get the error distribution of A. More specifically, we first calculate the error pattern probability from the bit flip probability; then, the distribution of A' is derived from the error pattern probability. In other words, we can use the following steps to get the error distribution in operand: **bit flip probability** $\rightarrow$ **error pattern probability** $\rightarrow$ **operand error probability**. Here, "bit flip probability" means the probability for $1 \rightarrow 0$

or $0 \rightarrow 1$. The "error pattern probability" tells us which set of bits have bit flip. Intuitively, error pattern is a mask where 1 denotes bit flip at this bit, while 0 not. "operand error probability" means the error distribution of one operand. In order to get this value, for each error pattern, we first apply one error pattern to each operand to get the bit flipped operand, then we subtract these two values to get the operand error.

The step from bit flip probability to error pattern probability is easy, since we only need to know the number of 1s and 0s in the error pattern.

When we calculate operand error probability from error pattern probability, we assume that operand A's value is uniform distribution. This assumption is not for the simplification of analysis. We can use other distribution types if those better represent the distribution of operand values. The application of one error pattern to these values will produce a new set of values. Under one error pattern, these new values also satisfy uniform distribution. By subtracting a new value from its corresponding original value, we get the error in operand. If we group the pairs of values causing the same error, we will see that all the groups have the same number of members. For example, if we assume the precision is 5, then there are 16 different values for operand; under one error pattern, if there are 4 different error values, then each of these four error values appear exactly four times. See Appendix for specific method used to calculate operand's error distribution.

2.2.1 Simulation of Error in Addition

The above analysis provides a much simplified base for simulating the error in the result of addition. The error is divided into 9 categories as shown in Table 2.3. We need to go over possible cases in the 9 categories in the simulation to get the overall distribution of error ($C' - C$). Our simulation algorithm does exactly the walk-through of the 9 categories. Its pseudo-code is shown in Figure 2.1. In the pseudo-code, d is defined as $E_A - E_B$, Portion_$\{C'_no_carry, C_no_carry\}$ means the probability that neither C' nor C has carry, Pr_Δ_A means the probability for the current error in A, and Pr_d means the probability for the current d.

for $d \leftarrow d_{min}$ to d_{max} **do**
  if $(d = 0)$ **then**
    for each Δ_A **do**
      $\Delta_C \leftarrow [\Delta_A \times 2^{-1} - 0.\underbrace{00\cdots00}_{p-2}1, \Delta_A \times 2^{-1} + 0.\underbrace{00\cdots00}_{p-2}1]$
      $\mathrm{Pr}\_\Delta_C \leftarrow \mathrm{Pr}\_\Delta_A \times \mathrm{Pr}\_d$
    end for
  end if
  if $(d > 0)$ **then**
    for each Δ_A **do**
      if $(C', C) \leftarrow (1.1, 1.1)$ **then**
        $\Delta_C \leftarrow [\Delta_A - 0.\underbrace{00\cdots00}_{p-2}1, \Delta_A + 0.\underbrace{00\cdots00}_{p-2}1]$
        $\mathrm{Pr}\_\Delta_C \leftarrow \mathrm{Portion}\_\{C'\_no\_carry, C\_no\_carry\} \times \mathrm{Pr}\_\Delta_A \times \mathrm{Pr}\_d$
      else if $(C', C) \leftarrow (1.2, 1.2)$ **then**
        $\Delta_C \leftarrow [\Delta_A \times 2^{-1} - 0.\underbrace{00\cdots00}_{p-2}1, \Delta_A \times 2^{-1} + 0.\underbrace{00\cdots00}_{p-2}1]$
        $\mathrm{Pr}\_\Delta_C \leftarrow \mathrm{Portion}\_\{C'\_carry, C\_carry\} \times \mathrm{Pr}\_\Delta_A \times \mathrm{Pr}\_d$
      else if $(C', C) \leftarrow (1.1, 1.2)$ **then**
        $\Delta_C \leftarrow [\Delta_A \times 2^{-1} - 0.\underbrace{00\cdots00}_{p-1}11, \Delta_A \times 2^{-1} + 0.\underbrace{00\cdots00}_{p-1}11]$
        $\mathrm{Pr}\_\Delta_C \leftarrow \mathrm{Portion}\_\{C'\_no\_carry, C\_carry\} \times \mathrm{Pr}\_\Delta_A \times \mathrm{Pr}\_d$
      else if $(C', C) \leftarrow (1.2, 1.1)$ **then**
        $\Delta_C \leftarrow [\Delta_A \times 2^{-1} - 0.\underbrace{00\cdots00}_{p-1}11, \Delta_A \times 2^{-1} + 0.\underbrace{00\cdots00}_{p-1}11]$
        $\mathrm{Pr}\_\Delta_C \leftarrow \mathrm{Portion}\_\{C'\_carry, C\_no\_carry\} \times \mathrm{Pr}\_\Delta_A \times \mathrm{Pr}\_d$
      end if
    end for
  end if
  if $(d < 0)$ **then**
    for each Δ_A **do**
      if $(C', C) \leftarrow (3.1, 3.1)$ **then**
        $\Delta_C \leftarrow [\Delta_A \times 2^{-d} - 0.\underbrace{00\cdots00}_{p-2}1, \Delta_A \times 2^{-d} + 0.\underbrace{00\cdots00}_{p-2}1]$
        $\mathrm{Pr}\_\Delta_C \leftarrow \mathrm{Portion}\_\{C'\_no\_carry, C\_no\_carry\} \times \mathrm{Pr}\_\Delta_A \times \mathrm{Pr}\_d$
      else if $(C', C) \leftarrow (3.2, 3.2)$ **then**
        $\Delta_C \leftarrow [\Delta_A \times 2^{-(d+1)} - 0.\underbrace{00\cdots00}_{p-2}1, \Delta_A \times 2^{-(d+1)} + 0.\underbrace{00\cdots00}_{p-2}1]$
        $\mathrm{Pr}\_\Delta_C \leftarrow \mathrm{Portion}\_\{C'\_carry, C\_carry\} \times \mathrm{Pr}\_\Delta_A \times \mathrm{Pr}\_d$
      else if $(C', C) \leftarrow (3.1, 3.2)$ **then**
        $\Delta_C \leftarrow [\Delta_A \times 2^{-(d+1)} - 0.\underbrace{00\cdots00}_{p-1}11, \Delta_A \times 2^{-(d+1)} + 0.\underbrace{00\cdots00}_{p-1}11]$
        $\mathrm{Pr}\_\Delta_C \leftarrow \mathrm{Portion}\_\{C'\_no\_carry, C\_carry\} \times \mathrm{Pr}\_\Delta_A \times \mathrm{Pr}\_d$
      else if $(C', C) \leftarrow (3.2, 3.1)$ **then**
        $\Delta_C \leftarrow [\Delta_A \times 2^{-(d+1)} - 0.\underbrace{00\cdots00}_{p-1}11, \Delta_A \times 2^{-(d+1)} + 0.\underbrace{00\cdots00}_{p-1}11]$
        $\mathrm{Pr}\_\Delta_C \leftarrow \mathrm{Portion}\_\{C'\_carry, C\_no\_carry\} \times \mathrm{Pr}\_\Delta_A \times \mathrm{Pr}\_d$
      end if
    end for
  end if
end for

Figure 2.1: Pseudo code of *addition*

2.3 Bit Flip Error Propagation in Subtraction

For subtraction, we still assume only one of the two operands has bit flip and bit flip only occurs in fraction part. So, C' is either $(A' - B)$ or $(A - B')$, while C is $(A - B)$ in both cases. Since we assume both operands (i.e., A and B) having same error distribution, the difference between errors coming from B and A is a sign. If we have known the expressions of the error in the result caused by A, then for B, we can get it by putting a minus sign before each expression for error in the result. So we only analyze the error coming from A. In this section, we first analyze the error caused by bit flip from A and then give an algorithm to simulate bit flip error propagation in subtraction.

First we illustrate the exact process of how C is computed using a standard floating-point format. To subtract A and B, first we need to align these two numbers according to their exponent values, that is, shifting left the operand with smaller exponent value by the difference between these two exponent values (equivalently, increase the exponent value). After getting the intermediate result of subtraction, we need to normalize it (equivalently, decrease the exponent value). The next step is to round the normalized result according to the rounding mode. Here we assume the rounding mode is round-to-nearest.

Hence we can get the expressions for C as shown in Table 2.4.

In Table 2.4, we assume the first non-zero bit is the k^{th} bit of the intermediate result (which is gotten by aligning A and B, then performing A minus B). In order to normalize this intermediate result, we need shift the binary point to right by k bits, equivalent to multiply it by 2^k.

Since the difference between C and C' comes from the value of one operand, we use A' for C' instead of A (same B for both C and C').

C' may have the expressions as shown in Table 2.5.

From Table 2.4 and Table 2.5, we know the pair of (C', C) may have the following different combinations:

- for $E_A > E_B$, (C', C) may be (1.1, 1.*), (1.2, 1.*), (1.3, 1.*).

- for $E_A = E_B$, there is only one case which is (2, 2).
- for $E_A < E_B$, (C', C) may be (3.1, 3.*), (3.2, 3.*), (3.3, 3.*).

(Note: in the above, * may be 1, 2, or 3.)

By combining the two tables for the expressions of C and C', we can get the expressions of the error in the result ($C' - C$) as shown in Table 2.6. In this table, we denote the number of bits has to be shifted for C is k, for C' is k'. The d in the table is defined as: 1) for (1.*, 1.*), $d = E_A - E_B$; and 2) for (3.*, 3.*), $d = E_B - E_A$. Δ_A is defined as $\Delta_A = 1.a'_1 a'_2 \cdots a'_{p-1} - 1.a_1 a_2 \cdots a_{p-1}$.

2.3.1 Simulation of Error in Subtraction

Table 2.6 represents the expressions of error in the subtraction operation in terms of the bit flip error in operand A. Next we can simulate the error in C by walking through all categories defined in that table. Here we need to discuss several special cases in the simulation.

If $d = E_A - E_B$ and when $d = 0$, the value of borrow ranges from 1 to $(p-1)$. An exception is that when the intermediate result is 0, it means there is no need to normalize at all. When $|d| = 1$, the value of borrow ranges from 0 to p. And when $|d| > 1$, the value of borrow is either 0 or 1. Here, the "borrow" is the number of bits that has to be shifted in order to normalize the intermediate result. For subtraction, the exponent value is always decreased. When $|d| = 0$ or 1, we need know the specific value of borrow to determine how to get the error. When $|d| > 1$, we can use the method similar to that in the analysis of addition to calculate the error. Putting every case together, we can get the algorithms for subtraction as shown in Figure 2.2. In the algorithm, Portion_$\{C'_no_borrow, C_no_borrow\}$ has the same meaning as in Section 2.2, Pr_Δ_A means the probability for the current error in A, and Pr_d means the probability for the current d.

for $d \leftarrow d_{min}$ to d_{max} **do**
  if $d = 0$ or $d = 1$ or $d = -1$ **then**
    //specific values of borrow is used to calculate the error according to Table 2.6
  end if
  if $d \geq 2$ **then**
    for each Δ_A **do**
      //first determine the borrow is 1 or 0, then according to Table 2.6 to select appropriate formula to perform calculation; then, use the similar way as in Section 2.2.1 to calculate the corresponding probability
    end for
  end if
  if $d \leq -2$ **then**
    for each Δ_A **do**
      //first determine the borrow is 1 or 0, then according to Table 2.6 to select appropriate formula to perform calculation; then, use the similar way as in Section 2.2.1 to calculate the corresponding probability
    end for
  end if
end for

Figure 2.2: Pseudo code of *subtraction*

Table 2.4: Expressions for $C\ (= A - B)$

1. $E_A > E_B$	1.1. $k > d$ $[1a_1a_2\cdots a_k.a_{k+1}\cdots a_{p-1} - 1b_1b_2\cdots b_{k-d}.b_{k-d+1}\cdots b_{p-1} -$ $0.\underbrace{00\cdots00}_{p-1}1, 1a_1a_2\cdots a_k.a_{k+1}\cdots a_{p-1} - 1b_1b_2\cdots b_{k-d}.b_{k-d+1}\cdots$ $b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A-k}$ 1.2. $k = d$ $[1a_1a_2\cdots a_k.a_{k+1}\cdots a_{p-1} - 1.b_1\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $1a_1a_2\cdots a_k.a_{k+1}\cdots a_{p-1} - 1.b_1\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A-k}$ 1.3. $k < d$ $[1a_1a_2\cdots a_k.a_{k+1}\cdots a_{p-1} - 0.\underbrace{00\cdots00}_{d-k-1}1b_1\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $1a_1a_2\cdots a_k.a_{k+1}\cdots a_{p-1} - 0.\underbrace{00\cdots00}_{d-k-1}1b_1\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A-k}$ $(d = E_A - E_B)$
2. $E_A = E_B$	$[1a_1a_2\cdots a_k.a_{k+1}\cdots a_{p-1} - 1b_1b_2\cdots b_k.b_{k+1}\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $1a_1a_2\cdots a_k.a_{k+1}\cdots a_{p-1} - 1b_1b_2\cdots b_k.b_{k+1}\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E-k}$ $(E = E_A = E_B)$
3. $E_A < E_B$	3.1. $k > d$ $[1a_1a_2\cdots a_{k-d}.a_{k-d+1}\cdots a_{p-1} - 1b_1b_2\cdots b_k.b_{k+1}\cdots b_{p-1} -$ $0.\underbrace{00\cdots00}_{p-1}1, 1a_1a_2\cdots a_{k-d}.a_{k-d+1}\cdots a_{p-1} - 1b_1b_2\cdots b_k.b_{k+1}\cdots$ $b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_B-k}$ 3.2. $k = d$ $[1.a_1\cdots a_{p-1} - 1b_1b_2\cdots b_k.b_{k+1}\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $1.a_1\cdots a_{p-1} - 1b_1b_2\cdots b_k.b_{k+1}\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_B-k}$ 3.3. $k < d$ $[0.\underbrace{00\cdots00}_{d-k-1}1a_1\cdots a_{p-1} - 1b_1b_2\cdots b_k.b_{k+1}\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $0.\underbrace{00\cdots00}_{d-k-1}1a_1\cdots a_{p-1} - 1b_1b_2\cdots b_k.b_{k+1}\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_B-k}$ $(d = E_B - E_A)$

Table 2.5: Expressions for C' $(= A' - B)$

1. $E_A > E_B$	1.1. $k > d$ $[1a'_1a'_2\cdots a'_k.a'_{k+1}\cdots a'_{p-1} - 1b_1b_2\cdots b_{k-d}.b_{k-d+1}\cdots b_{p-1} -$ $0.\underbrace{00\cdots00}_{p-1}1, 1a'_1a'_2\cdots a'_k.a'_{k+1}\cdots a'_{p-1} - 1b_1b_2\cdots b_{k-d}.b_{k-d+1}\cdots$ $b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A-k}$ 1.2. $k = d$ $[1a'_1a'_2\cdots a'_k.a'_{k+1}\cdots a'_{p-1} - 1.b_1\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $1a'_1a'_2\cdots a'_k.a'_{k+1}\cdots a'_{p-1} - 1.b_1\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A-k}$ 1.3. $k < d$ $[1a'_1a'_2\cdots a'_k.a'_{k+1}\cdots a'_{p-1} - 0.\underbrace{00\cdots00}_{d-k-1}1b_1\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $1a'_1a'_2\cdots a'_k.a'_{k+1}\cdots a'_{p-1} - 0.\underbrace{00\cdots00}_{d-k-1}1b_1\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A-k}$ $(d = E_A - E_B)$
2. $E_A = E_B$	$[1a'_1a'_2\cdots a'_k.a'_{k+1}\cdots a'_{p-1} - 1b_1b_2\cdots b_k.b_{k+1}\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $1a'_1a'_2\cdots a'_k.a'_{k+1}\cdots a'_{p-1} - 1b_1b_2\cdots b_k.b_{k+1}\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E-k}$ $(E = E_A = E_B)$
3. $E_A < E_B$	3.1. $k > d$ $[1a'_1a'_2\cdots a'_{k-d}.a'_{k-d+1}\cdots a'_{p-1} - 1b_1b_2\cdots b_k.b_{k+1}\cdots b_{p-1} -$ $0.\underbrace{00\cdots00}_{p-1}1, 1a'_1a'_2\cdots a'_{k-d}.a'_{k-d+1}\cdots a'_{p-1} - 1b_1b_2\cdots b_k.b_{k+1}\cdots$ $b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_B-k}$ 3.2. $k = d$ $[1.a'_1\cdots a'_{p-1} - 1b_1b_2\cdots b_k.b_{k+1}\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $1.a'_1\cdots a'_{p-1} - 1b_1b_2\cdots b_k.b_{k+1}\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_B-k}$ 3.3. $k < d$ $[0.\underbrace{00\cdots00}_{d-k-1}1a'_1\cdots a'_{p-1} - 1b_1b_2\cdots b_k.b_{k+1}\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $0.\underbrace{00\cdots00}_{d-k-1}1a'_1\cdots a'_{p-1} - 1b_1b_2\cdots b_k.b_{k+1}\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_B-k}$ $(d = E_B - E_A)$

Table 2.6: Error of Subtraction Operation ($C' - C$)

(1.*., 1.*.)	$[\Delta_A \times 2^{min\{k,k'\}} - 0.\underbrace{00\cdots00}_{p-1}1 - 0.\underbrace{00\cdots00}_{con}1,$ $\Delta_A \times 2^{min\{k,k'\}} + 0.\underbrace{00\cdots00}_{p-1}1 + 0.\underbrace{00\cdots00}_{con}1] \times 2^{E_A - min\{k,k'\}}$
(2, 2)	$[\Delta_A \times 2^{min\{k,k'\}} - 0.\underbrace{00\cdots00}_{p-1}1 - 0.\underbrace{00\cdots00}_{con}1,$ $\Delta_A \times 2^{min\{k,k'\}} + 0.\underbrace{00\cdots00}_{p-1}1 + 0.\underbrace{00\cdots00}_{con}1] \times 2^{E - min\{k,k'\}}$
(3.*, 3.*)	$[\Delta_A \times 2^{min\{k,k'\}} \div 2^d - 0.\underbrace{00\cdots00}_{p-1}1 - 0.\underbrace{00\cdots00}_{con}1,$ $\Delta_A \times 2^{min\{k,k'\}} \div 2^d + 0.\underbrace{00\cdots00}_{p-1}1 + 0.\underbrace{00\cdots00}_{con}1] \times 2^{E_B - min\{k,k'\}}$
	Note: $con = p - 1 + max\{k,k'\} - min\{k,k'\}$

2.4 Bit Flip Error Propagation in Multiplication

Because multiplication is commutative, we only need consider bit flip error in one of the two operands. Suppose it is A that has bit flip error, then C' is $A' \times B$, while C is $A \times B$. In this section, we first analyze bit flip error in multiplication; then give an algorithm to calculate bit flip error propagation in multiplication.

When two floating-point numbers are multiplied, we first multiply two significands, then add two exponents, and lastly shift the binary point in order to normalize the result. Moreover, the carry is either zero or one. Here, carry means the number of bits that has to be shifted in order to normalize the intermediate result.

The expressions for C are shown in Table 2.7.

The expressions for C' are shown in Table 2.8.

Table 2.7: Expressions for C $(= A \times B)$

1. No carry:	$[1.a_1a_2\cdots a_{p-1} \times 1.b_1b_2\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $1.a_1a_2\cdots a_{p-1} \times 1.b_1b_2\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A+E_B}$
2. Yes carry:	$[1.a_1a_2\cdots a_{p-1} \times 1.b_1b_2\cdots b_{p-1} \times 2^{-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $1.a_1a_2\cdots a_{p-1} \times 1.b_1b_2\cdots b_{p-1} \times 2^{-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A+E_B+1}$

Table 2.8: Expressions for C' $(= A' \times B)$

1. No carry:	$[1.a'_1a'_2\cdots a'_{p-1} \times 1.b_1b_2\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $1.a'_1a'_2\cdots a'_{p-1} \times 1.b_1b_2\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A+E_B}$
2. Yes carry:	$[1.a'_1a'_2\cdots a'_{p-1} \times 1.b_1b_2\cdots b_{p-1} \times 2^{-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $1.a'_1a'_2\cdots a'_{p-1} \times 1.b_1b_2\cdots b_{p-1} \times 2^{-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A+E_B+1}$

Based on the expressions for C and C', we know (C', C) may be (1, 1), (2, 2), (1, 2), or (2, 1). Thus, we can get the error in result as shown in Table 2.9. In Table 2.9, b means $1.b_1b_2\cdots b_{p-1}$. Unlike addition and subtraction, Table 2.9 shows that $C'-C$ is a function of the error in A and the value of B.

Table 2.9: Error of Multiplication Operation $(C'-C)$

(1, 1)	$[\Delta_A \times b - 0.\underbrace{00\cdots00}_{p-2}1, \Delta_A \times b + 0.\underbrace{00\cdots00}_{p-2}1] \times 2^{E_A+E_B}$
(2, 2)	$[\Delta_A \times b \times 2^{-1} - 0.\underbrace{00\cdots00}_{p-2}1, \Delta_A \times b \times 2^{-1} + 0.\underbrace{00\cdots00}_{p-2}1] \times 2^{E_A+E_B+1}$
(1, 2)	$[\Delta_A \times b \times 2^{-1} - 0.\underbrace{00\cdots00}_{p-1}11, \Delta_A \times b \times 2^{-1} + 0.\underbrace{00\cdots00}_{p-1}11] \times 2^{E_A+E_B+1}$
(2, 1)	$[\Delta_A \times b \times 2^{-1} - 0.\underbrace{00\cdots00}_{p-1}11, \Delta_A \times b \times 2^{-1} + 0.\underbrace{00\cdots00}_{p-1}11] \times 2^{E_A+E_B+1}$

2.4.1 Simulation of Error in Multiplication

Based on Table 2.9, to determine the error in result, we need know not only the error in operand A, but also the value of B. The basic idea of simulating the bit flip error propagation in multiplication is the same as that of addition and subtraction, that is, to walk through all categories defined in Table 2.9. But since multiplying two numbers does not involve the difference between two exponents, there is no need to walk through different values for exponent difference (d) as in addition and subtraction. The algorithm that can be used is shown in Figure 2.3. Here, we use b to denote $1.b_1b_2\cdots b_{p-1}$. Portion_$\{C'_no_carry, C_no_carry\}$ has the same meaning as in Section 2.2.1.

for each B **do**
  for each Δ_A **do**
    if $(C', C) \leftarrow (1,1)$ **then**
      $\Delta_C \leftarrow [\Delta_A \times b - 0.\underbrace{00\cdots00}_{p-2}1, \Delta_A \times b + 0.\underbrace{00\cdots00}_{p-2}1]$
      $\Pr\_\Delta_C \leftarrow \text{Portion\_}\{C'\_no\_carry, C\_no\_carry\} \times \Pr\_\Delta_A \times \Pr\_B$;
    else if $(C', C) \leftarrow (2,2)$ **then**
      $\Delta_C \leftarrow [\Delta_A \times b \times 2^{-1} - 0.\underbrace{00\cdots00}_{p-2}1, \Delta_A \times b \times 2^{-1} + 0.\underbrace{00\cdots00}_{p-2}1]$
      $\Pr\_\Delta_C \leftarrow \text{Portion\_}\{C'\_carry, C\_carry\} \times \Pr\_\Delta_A \times \Pr\_B$;
    else if $(C', C) \leftarrow (1,2)$ **then**
      $\Delta_C \leftarrow [\Delta_A \times b \times 2^{-1} - 0.\underbrace{00\cdots00}_{p-1}11, \Delta_A \times b \times 2^{-1} + 0.\underbrace{00\cdots00}_{p-1}11]$
      $\Pr\_\Delta_C \leftarrow \text{Portion\_}\{C'\_no\_carry, C\_carry\} \times \Pr\_\Delta_A \times \Pr\_B$;
    else if $(C', C) \leftarrow (2,1)$ **then**
      $\Delta_C \leftarrow [\Delta_A \times b \times 2^{-1} - 0.\underbrace{00\cdots00}_{p-1}11, \Delta_A \times b \times 2^{-1} + 0.\underbrace{00\cdots00}_{p-1}11]$
      $\Pr\_\Delta_C \leftarrow \text{Portion\_}\{C'\_carry, C\_no\_carry\} \times \Pr\_\Delta_A \times \Pr\_B$;
    end if
  end for
end for

Figure 2.3: Pseudo code of *Multiplication*

2.5 Bit Flip Error Propagation in Division

Division is not commutative. Therefore, the model of the bit flip error propagation needs to distinguish two cases: $C' = A' \div B$ and $C' = A \div B'$, while C is $A \div B$. In this section, we first analyze the error caused by operand A and give its corresponding simulation algorithm to calculate result error $(C' - C)$. Then, we analyze the error caused by operand B and give an algorithm to simulate such errors.

2.5.1 Error from Operand A

The division of two floating-point numbers is also a multi-step process. The two significands are firstly divided; then two exponents subtracted; and lastly if the intermediate result is not in normalized form, the binary point is shifted to normalize the result. Moreover, after dividing two significands, the borrow is either zero or one. Here, the "borrow" means the number of bits that has to be shifted in order to normalize the intermediate result. Here we omit the details of the modeling, which is similar to the cases for addition, and directly present the analysis result.

The expressions for C are shown in Table 2.10.

The expressions for C' are shown in Table 2.11.

Table 2.10: Expressions for C $(= A \div B)$

1. No borrow:	$[1.a_1a_2\cdots a_{p-1} \div 1.b_1b_2\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $1.a_1a_2\cdots a_{p-1} \div 1.b_1b_2\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A - E_B}$
2. Yes borrow:	$[2 \times 1.a_1a_2\cdots a_{p-1} \div 1.b_1b_2\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $2 \times 1.a_1a_2\cdots a_{p-1} \div 1.b_1b_2\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A - E_B - 1}$

Based on the expressions for C and C', we know (C', C) may be (1, 1), (2, 2), (1, 2), or (2, 1). Thus, we can get the error in result as shown in Table 2.12. In this Table,

Table 2.11: Expressions for C' $(= A' \div B)$

1. No borrow:	$[1.a'_1a'_2\cdots a'_{p-1} \div 1.b_1b_2\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $1.a'_1a'_2\cdots a'_{p-1} \div 1.b_1b_2\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A-E_B}$
2. Yes borrow:	$[2\times 1.a'_1a'_2\cdots a'_{p-1} \div 1.b_1b_2\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $2\times 1.a'_1a'_2\cdots a'_{p-1} \div 1.b_1b_2\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A-E_B-1}$

b means $1.b_1b_2\cdots b_{p-1}$. Table 2.12 shows that the error in A and the value of B together determine the error in result.

Table 2.12: Error of Division Operation $(C'-C)(1)$

(1, 1)	$[\Delta_A \div b - 0.\underbrace{00\cdots00}_{p-2}1, \Delta_A \div b + 0.\underbrace{00\cdots00}_{p-2}1] \times 2^{E_A-E_B}$
(2, 2)	$[2\times\Delta_A \div b - 0.\underbrace{00\cdots00}_{p-2}1, 2\times\Delta_A \div b + 0.\underbrace{00\cdots00}_{p-2}1] \times 2^{E_A-E_B-1}$
(1, 2)	$[\Delta_A \div b - 0.\underbrace{00\cdots00}_{p-1}11, \Delta_A \div b + 0.\underbrace{00\cdots00}_{p-1}11] \times 2^{E_A-E_B-1}$
(2, 1)	$[\Delta_A \div b - 0.\underbrace{00\cdots00}_{p-1}11, \Delta_A \div b + 0.\underbrace{00\cdots00}_{p-1}11] \times 2^{E_A-E_B-1}$

Consequently, the algorithm to simulate the error propagation in $C' = A'/B$ is developed from Table 2.12 in a similar way, as shown in Figure 2.4. In the pseudo code, Portion_$\{C'_no_borrow, C_no_borrow\}$ has the same meaning as in Section 2.2.1.

2.5.2 Error from Operand B

In this case, C still has the same expressions as in section 2.5.1. The expression of C' is different, however. We define B' as:

$$B' = 1.b'_1b'_2\cdots b'_{p-1} \times 2^{E_B}$$

where the exponent is the same with B's (E_B), and $\{b'_1, b'_2, \cdots, b'_{p-1}\} \in \{0, 1\}$. Saving the details of modeling, C' have the expressions as shown in Table 2.13.

for each B **do**
  for each Δ_A **do**
    if $(C', C) \leftarrow (1,1)$ **then**
      $\Delta_C \leftarrow [\Delta_A \div 1.b_1b_2\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-2}1, \Delta_A \div 1.b_1b_2\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-2}1]$
      Pr_$\Delta_C \leftarrow$ Portion_$\{C'$_*no_borrow*$, C$_*no_borrow*$\} \times$ Pr_$\Delta_A \times$ Pr_B;
    else if $(C', C) \leftarrow (2,2)$ **then**
      $\Delta_C \leftarrow [2 \times \Delta_A \div 1.b_1b_2\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-2}1, 2 \times \Delta_A \div 1.b_1b_2\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-2}1]$
      Pr_$\Delta_C \leftarrow$ Portion_$\{C'$_*borrow*$, C$_*borrow*$\} \times$ Pr_$\Delta_A \times$ Pr_B;
    else if $(C', C) \leftarrow (1,2)$ **then**
      $\Delta_C \leftarrow [\Delta_A \div 1.b_1b_2\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}11, \Delta_A \div 1.b_1b_2\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}11]$
      Pr_$\Delta_C \leftarrow$ Portion_$\{C'$_*no_borrow*$, C$_*borrow*$\} \times$ Pr_$\Delta_A \times$ Pr_B;
    else if $(C', C) \leftarrow (2,1)$ **then**
      $\Delta_C \leftarrow [\Delta_A \div 1.b_1b_2\cdots b_{p-1} - 0.\underbrace{00\cdots00}_{p-1}11, \Delta_A \div 1.b_1b_2\cdots b_{p-1} + 0.\underbrace{00\cdots00}_{p-1}11]$
      Pr_$\Delta_C \leftarrow$ Portion_$\{C'$_*borrow*$, C$_*no_borrow*$\} \times$ Pr_$\Delta_A \times$ Pr_B;
    end if
  end for
end for

Figure 2.4: Pseudo code of *Division(1)*

Based on the expressions for C and C', we know (C', C) may be (1, 1), (2, 2), (1, 2), (2, 1). Thus, we can get the error in the result as shown in Table 2.14.

From Table 2.14, in order to determine the error in result, we need to know the error in B, as well as the values of both A and B. The algorithm we can use is shown in Figure 2.5.

Table 2.13: Expressions for C' $(= A \div B')$

1. No borrow:	$[1.a_1a_2\cdots a_{p-1} \div 1.b'_1b'_2\cdots b'_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $1.a_1a_2\cdots a_{p-1} \div 1.b'_1b'_2\cdots b'_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A-E_B}$
2. Yes borrow:	$[2 \times 1.a_1a_2\cdots a_{p-1} \div 1.b'_1b'_2\cdots b'_{p-1} - 0.\underbrace{00\cdots00}_{p-1}1,$ $2 \times 1.a_1a_2\cdots a_{p-1} \div 1.b'_1b'_2\cdots b'_{p-1} + 0.\underbrace{00\cdots00}_{p-1}1] \times 2^{E_A-E_B-1}$

for each A **do**
  for each B **do**
    for each B' **do**
      mask = B XOR B'; //get the error pattern
      //For the four cases as shown in Table 2.14, first, according to A, B, and B', select appropriate formula; then compute the error in C;
      //lastly, according to the formula below to calculate the corresponding probability.
      Pr_$\Delta_C \leftarrow$ Pr_$A \times$ Pr_$B \times$ Pr_*mask*;
    end for
  end for
end for

Figure 2.5: Pseudo code of *Division(2)*

Table 2.14: Error of Division Operation $(C' - C)(2)$

(1, 1)	$[1.a_1 \cdots a_{p-1} \times (\frac{1}{1.b'_1 \cdots b'_{p-1}} - \frac{1}{1.b_1 \cdots b_{p-1}}) - 0.\underbrace{00 \cdots 00}_{p-2}1,$ $1.a_1 \cdots a_{p-1} \times (\frac{1}{1.b'_1 \cdots b'_{p-1}} - \frac{1}{1.b_1 \cdots b_{p-1}}) + 0.\underbrace{00 \cdots 00}_{p-2}1] \times 2^{E_A - E_B}$
(2, 2)	$[2 \times 1.a_1 \cdots a_{p-1} \times (\frac{1}{1.b'_1 \cdots b'_{p-1}} - \frac{1}{1.b_1 \cdots b_{p-1}}) - 0.\underbrace{00 \cdots 00}_{p-2}1,$ $2 \times 1.a_1 \cdots a_{p-1} \times (\frac{1}{1.b'_1 \cdots b'_{p-1}} - \frac{1}{1.b_1 \cdots b_{p-1}}) + 0.\underbrace{00 \cdots 00}_{p-2}1] \times 2^{E_A - E_B - 1}$
(1, 2)	$[1.a_1 \cdots a_{p-1} \times (\frac{1}{1.b'_1 \cdots b'_{p-1}} - \frac{1}{1.b_1 \cdots b_{p-1}}) - 0.\underbrace{00 \cdots 00}_{p-1}11,$ $1.a_1 \cdots a_{p-1} \times (\frac{1}{1.b'_1 \cdots b'_{p-1}} - \frac{1}{1.b_1 \cdots b_{p-1}}) + 0.\underbrace{00 \cdots 00}_{p-1}11] \times 2^{E_A - E_B}$
(2, 1)	$[1.a_1 \cdots a_{p-1} \times (\frac{1}{1.b'_1 \cdots b'_{p-1}} - \frac{1}{1.b_1 \cdots b_{p-1}}) - 0.\underbrace{00 \cdots 00}_{p-1}11,$ $1.a_1 \cdots a_{p-1} \times (\frac{1}{1.b'_1 \cdots b'_{p-1}} - \frac{1}{1.b_1 \cdots b_{p-1}}) + 0.\underbrace{00 \cdots 00}_{p-1}11] \times 2^{E_A - E_B}$

Chapter 3

RESULTS

The results mainly consist of two parts. The first part is the error propagation for the four basic arithmetic operations; the second part is the error tolerance that is empirically computed from the simulation. In addition, we give the simulation result between precision and threshold.

3.1 Error Propagation Results

In Figures 3.1 to 3.6, the X-axis represents the error in the result $(C' - C)$. This axis is symmetric around zero. Because the absolute values of the error results are very small, to show the values clearly, we show the corresponding normalized decimal value of the error. For example, if precision is p , then we scale up all error values by 2^{p-1}. For example, if $p = 5$ and $(C' - C)$ is 0.0010_{binary}, the corresponding normalized decimal value is 2, which is how the error value is shown in the Figures. The Y-axis represents error probability.

In Figure 3.7, the X-axis means precision which is the number of bits in significand. The Y-axis is the upper bound for certain threshold. As in Figures 3.1 to 3.6, the upper bound is normalized decimal value.

3.2 Addition

We simulate the bit flip error propagation in addition using the following settings: 1. the precision is 8; 2. the exponent changes from 1 to 16; 3. simulate four different single bit-flip probability (0.1, 0.01, 0.001, and 0.0001). Here, we assume the exponent distribution is uniform.

Figure 3.1 shows the error distribution based on the above parameters. In this figure, we can see that the error values at or near zero have much larger probability and the curve has some small peaks on both sides. If we try to get the accumulated error probability and set the threshold of error tolerance to be 0.9, then for 0.1, the interval is $[-32,32]$ ($[-0.0100000,0.0100000]_{binary}$); for 0.01, the interval is $[-1,1]$ ($[-0.0000001,0.0000001]_{binary}$); for 0.001, the interval is $[-1,1]$ ($[-0.0000001$, $0.0000001]_{binary}$); for 0.0001, the interval is $[-1,1]$ ($[-0.0000001,0.0000001]_{binary}$).

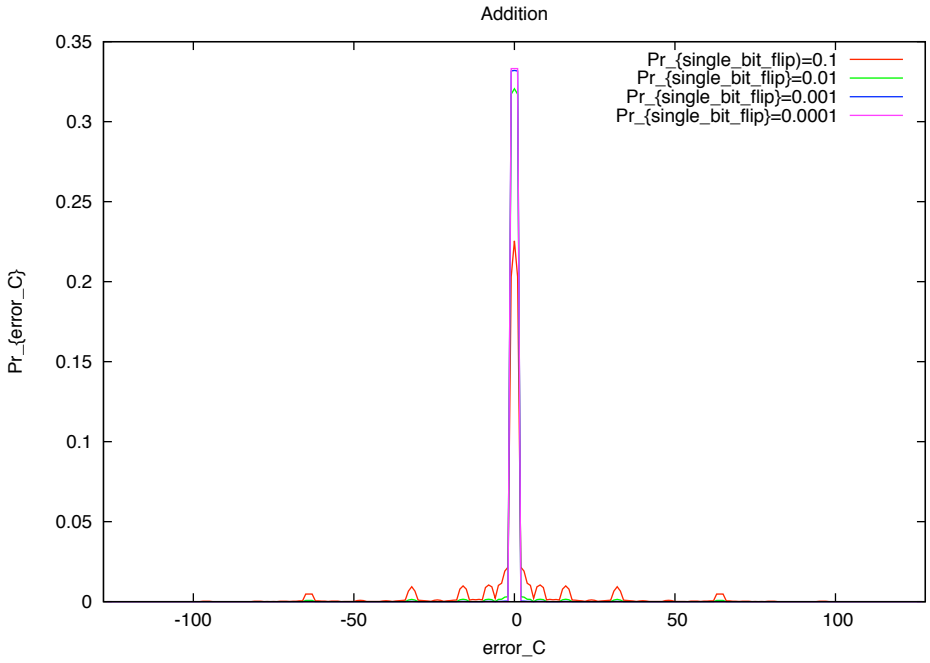

Figure 3.1: The simulation of error propagation in Addition: $A+B \rightarrow A'+B$

3.3 Subtraction

The settings for the simulation of subtraction operation are: 1. the precision is 8; 2. the exponent ranges from 1 to 16; 3. four different single bit-flip probability (0.1, 0.01, 0.001, 0.0001). Still, we assume the exponent distribution is uniform.

Figure 3.2 shows the error distribution if bit flip happens in A, and Figure 3.3 shows the case if the error comes from B. Both figures have some small peaks on both sides and the error values at or near zero have much greater probability. If we try to get the accumulated error probability and set the threshold of error tolerance to be 0.9, then for both figures, we have: for 0.1, the interval is $[-31,31]$ ($[-0.0011111,0.0011111]_{binary}$); for 0.01, the interval is $[-1,1]$ ($[-0.0000001,0.0000001]_{binary}$); for 0.001, the interval is $[-1,1]$ ($[-0.0000001,0.0000001]_{binary}$); for 0.0001, the interval is $[-1,1]$ ($[-0.0000001, 0.0000001]_{binary}$).

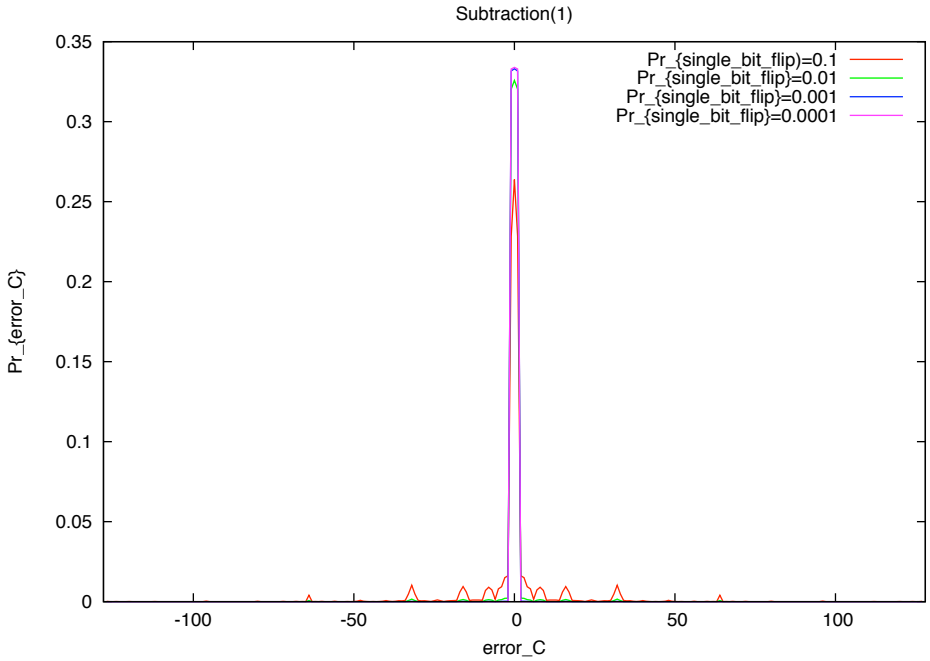

Figure 3.2: The simulation of error propagation in Subtraction: $A - B \rightarrow A' - B$

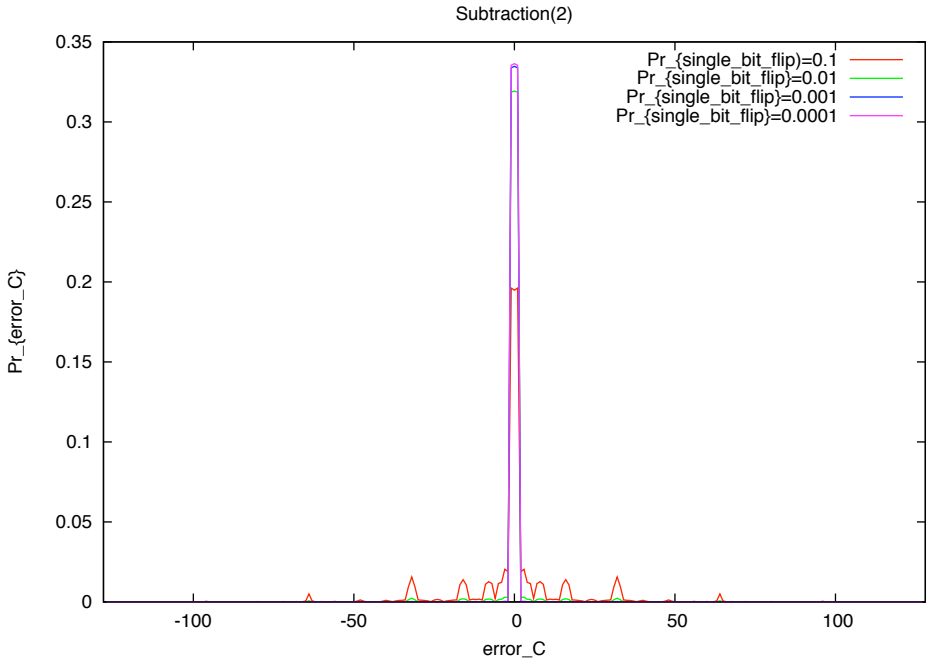

Figure 3.3: The simulation of error propagation in Subtraction: $A - B \rightarrow A - B'$

3.4 Multiplication

For multiplication, we provide the following settings: 1. the precision is 8; 2. four different single bit-flip probability (0.1, 0.01, 0.001, 0.0001). We still assume a uniform exponent distribution.

Figure 3.4 is the error distribution in multiplication. In this figure, we can see that error values at or near zero have much greater probability, which is similar to the other basic operations. However, there are no small peaks on the "wings" of either side. To get the accumulated error probability, if the threshold is 0.9, then for 0.1, the interval is $[-38, 38]$ ($[-0.0100110, 0.0100110]_{binary}$); for 0.01, the interval is $[-1, 1]$ ($[-0.0000001, 0.0000001]_{binary}$); for 0.001, the interval is $[-1, 1]$ ($[-0.0000001, 0.0000001]_{binary}$); for 0.0001, the interval is $[-1, 1]$ ($[-0.0000001, 0.0000001]_{binary}$).

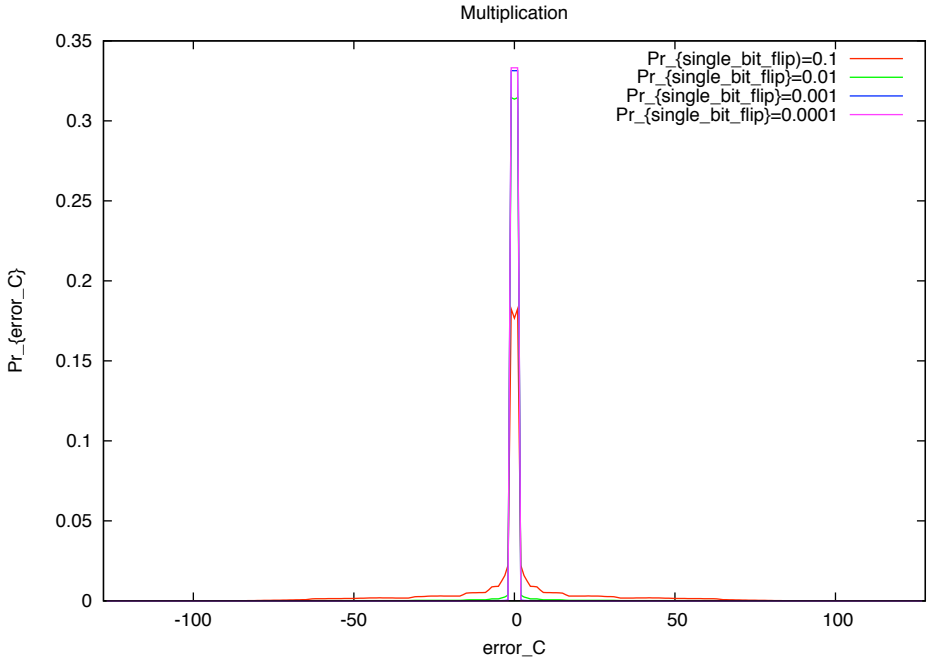

Figure 3.4: The simulation of error propagation in Multiplication: $A \times B \rightarrow A' \times B$

3.5 Division

For division, we use the following settings: 1. the precision is 8; 2. four different single bit-flip probability (0.1, 0.01, 0.001, 0.0001). The exponent distribution is uniform.

Figure 3.5 shows the error distribution if error comes from A, and Figure 3.6 for the error distribution if B has bit flip. Both figures have the maximal appear at or near zero and no small peaks on both sides. For the accumulated error probability, if the threshold is 0.9, then for both figures, we have: for 0.1, the interval is $[-38, 38]$ ($[-0.0100110, 0.0100110]_{binary}$); for 0.01, the interval is $[-1, 1]$ ($[-0.0000001, 0.0000001]_{binary}$); for 0.001, the interval is $[-1, 1]$ ($[-0.0000001, 0.0000001]_{binary}$); for 0.0001, the interval is $[-1, 1]$ ($[-0.0000001, 0.0000001]_{binary}$).

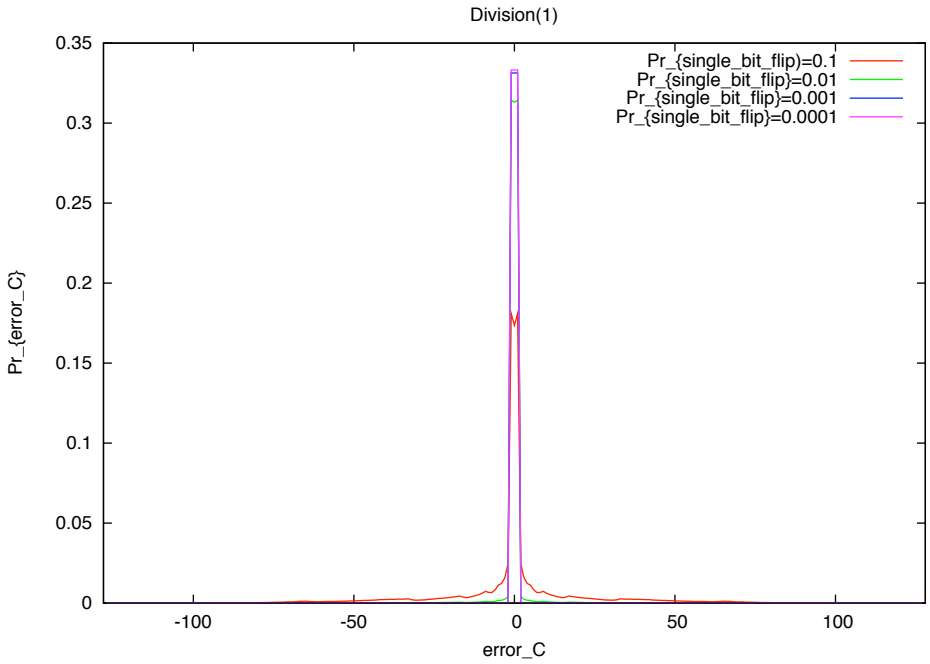

Figure 3.5: The simulation of error propagation in Division: $A \div B \rightarrow A' \div B$

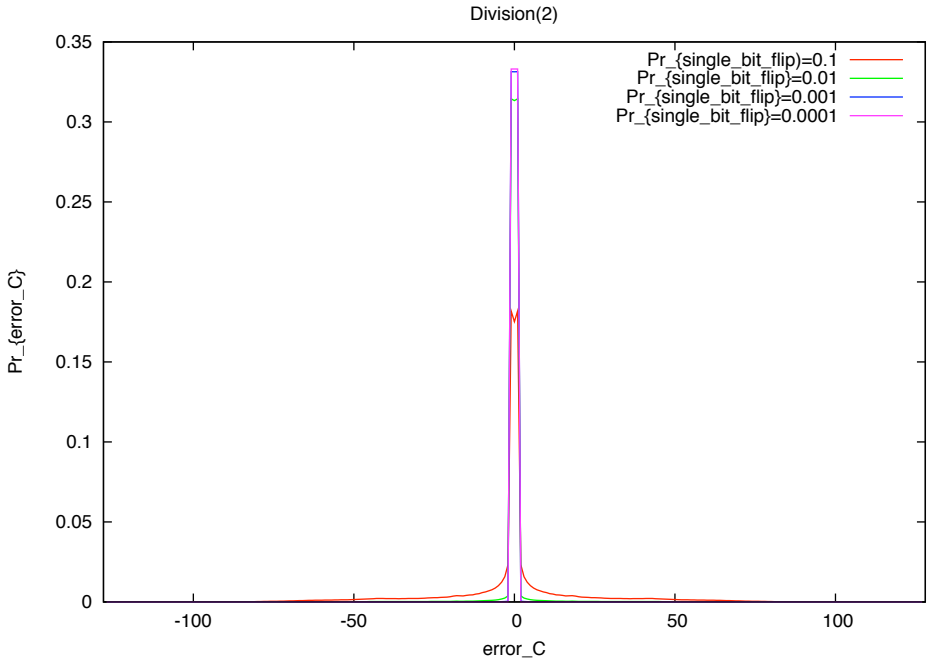

Figure 3.6: The simulation of error propagation in Division: $A \div B \rightarrow A \div B'$

3.6 Precision V.S. Threshold

In previous simulations, we fix the precision to be 8 and then get the threshold of tolerance. In this part, we show the simulation result for different precisions. Due to the similarity among the four arithmetic operations, we only show the result of multiplication. Moreover, we only show the upper bound because the curve is symmetric around 0.

Figure 3.7 shows the result of simulation. We can see that the relation between precision and upper bound is nearly exponential. Using this result, we can extrapolate the threshold for even bigger precision based on smaller precision. The reason why it exhibits exponential growth is because in our method, we use error pattern (mask) to get the operand error distribution, and then calculate the result error distribution from simulation. If the precision is increased by 1, the error range of operand is doubled which means the result error range is also doubled.

On the other hand, if we want to get Pr_$\{mask = 0.01\}$, then for precision = 3, Pr_$\{mask = 0.01\}$ = Pr_$\{mask = 0.01\}$. However, when precision = 4, Pr_$\{mask = 0.01\}$ = Pr_$\{mask = 0.010\}$ + Pr_$\{mask = 0.011\}$. It means if the precision increases by one, the probability of one error pattern will be propagated to at least two adjacent new error patterns. This propagation is disjoint which means any new error patterns will not get probability from more than one old error pattern.

Another observation is that the operand error values generated by applying masks to operands is not greater than the mask value, which means bigger masks may generate small error values, but smaller masks cannot generate bigger error values. So as precision increases, the probability of smaller value grows faster than bigger value's. The implication is that if precision is increased by one, we need to double the number of masks to get the same accumulated probability. But since bigger masks can generate smaller error values, the probability has the tendency to go to smaller error values. Thus, the upper bound is nearly doubled as precision increased by one.

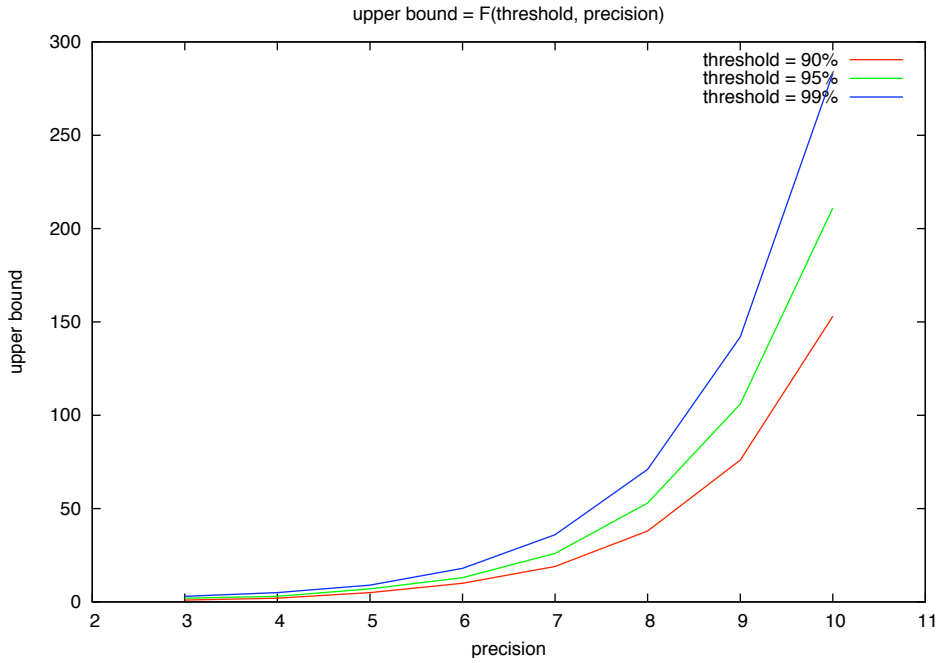

Figure 3.7: Upper bound = F(precision)

CONCLUSIONS

We propose a novel technique to evaluate the impact of bit flip caused by soft error on floating-point operations. Furthermore, we use empirical simulation to provide a quantitative model of what kinds of soft error can be tolerated. Our study is motivated by the observation that floating-point numbers are discretely represented and thus are inherently inaccurate, and that the semantics of floating-point programs usually can accept errors within a predefined range. We first build an analytic model that describes the error in the result of floating-point operations as a function of the bit flip error. Moreover, we use such functions to simulate the error propagation for all possible input values. Our result shows that for the four basic floating-point operations, there indeed exists an inherent error tolerance and we quantify such error propagation and tolerance.

There are a number of potential works to be done in the future. First, what we have considered is restricted to single operand and single floating-point operation bit flip error analysis. We need to extend our work to both operands and multiple floating-point operations, which is more general. Based on the current model, we can extend the work to both operands and more floating point operations straightforwardly. However, the cost would be huge. The cost comes as the performance. For example, if we want to consider two continual floating point operations, then the running time would double. So based on the analysis model we have now, we need to figure out a better way to extend the current work to a more general way.

Second, in our current model, we restrict the bit flip to the fraction part. But actually, bit flip may occur anywhere, i.e., exponent part and sign bit. If it happened in exponent part, then it would bring more influence than fraction part. To extend the model

to deal with exponent part, the cases we need consider are increased, since we need align the two numbers before subtracting them to get the error, and how to align is relative to the difference between two exponents.

Appendix

CALCULATION OF OPERAND'S ERROR DISTRIBUTION

This appendix shows how to calculate operand's error distribution. The idea has been described in section 2.2. Here is the specific method used to do such calculation. In the following sample code, num_mask means the number of error patterns (masks), error_pr means the probabilities of masks, and error_A_pr means the probabilities of operand's errors.

```
for(i=1; i<num_mask; i++){
    m=mask[i];
    z=0;
    x=1;

    //determine the # of different error values
    for(j=0; j<p-1; j++){
        x=1<<j;
        //compute the # of 1s in mask[i]
        if(m&x)z++;
    }

    //compute the # of different error values contained in this mask
    n=1<<z;

    //determine the error values and assign corresponding probability
    for(j=1; j<num_mask; j++){
        s=m&j;
        if(j==s){
            r = j<<1;
            //corresponding error value
```

```
            x = m - r;
            y = x + num_mask - 1;
            error_A_pr[y] += mask_pr[i]/n;
        }
    }
    error_A_pr[i+num_mask-1]+=mask_pr[i]/n;
}

error_A_pr[num_mask-1]=1;
for(i=0;i<p-1;i++)error_A_pr[num_mask-1]*=(1-pr);
```

BIBLIOGRAPHY

[1] Shekhar Borkar, Tanay Karnik, Siva Narendra, Jim Tschanz, Ali Keshavarzi, and Vivek De. Parameter variations and impact on circuits and microarchitecture. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 338–342, New York, NY, USA, 2003. ACM.

[2] R.C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on*, 5(3):305 – 316, sept. 2005.

[3] S. Mitra, Ming Zhang, T.M. Mak, N. Seifert, V. Zia, and Kee Sup Kim. Logic soft errors: a major barrier to robust platform design. In *Test Conference, 2005. Proceedings. ITC 2005. IEEE International*, pages 10 pp. –696, nov. 2005.

[4] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389–398, Washington, DC, USA, 2002. IEEE Computer Society.

[5] Nirmal R. Saxena, Santiago Fernandez-Gomez, Wei-Je Huang, Subhasish Mitra, Shu-Yi Yu, and Edward J. McCluskey. Dependable computing and online testing in adaptive and configurable systems. *IEEE Des. Test*, 17(1):29–41, 2000.

[6] Y.C. Yeh. Triple-triple redundant 777 primary flight computer. In *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, volume 1, pages 293 –307 vol.1, feb 1996.

[7] Y.C. Yeh. Design considerations in boeing 777 fly-by-wire computers. In *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*, pages 64 –72, nov 1998.

[8] Rajesh Venkatasubramanian, J.P. Hayes, and B.T. Murray. Low-cost on-line fault detection using control flow assertions. In *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, pages 137 – 143, july 2003.

[9] N. Oh, P.P. Shirvani, and E.J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *Reliability, IEEE Transactions on*, 51(1):63 –75, mar 2002.

[10] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. Swift: software implemented fault tolerance. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 243 – 254, march 2005.

[11] Jon G. Rokne. Interval arithmetic and interval analysis: an introduction. pages 1–22, 2001.

[12] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.