IEEE FLOATING-POINT EXTENSION FOR MANAGING ERROR USING RESIDUAL REGISTERS

By

ALEX UNDERWOOD

Bachelor of Science in Computer Engineering
Oklahoma State University
Stillwater, Oklahoma
2018

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2019

IEEE FLOATING-POINT EXTENSION FOR MANAGING ERROR
USING RESIDUAL REGISTERS

ACKNOWLEDGMENTS

I would like to sincerely thanks my adviser Dr. James E. Stine, Jr., for the tremendous amount of time and support he has given me during this research.

I would like to thank Dr. Keith Teague and Dr. Weili Zhang for serving as my committee members and always being available for assistance when I needed it.

I would like to express my thanks and appreciation to my parents, Michael and Deborah Underwood, for their love and support through all these years.

I would like to extend my sincere thanks to my brother, Tristan Underwood, who has been there to help and encourage me throughout my entire academic journey.

Name: ALEX UNDERWOOD

Date of Degree: DECEMBER, 2019

Title of Study: IEEE FLOATING-POINT EXTENSION FOR MANAGING ERROR USING RESIDUAL REGISTERS

Major Field: ELECTRICAL ENGINEERING

Abstract: This thesis discusses modifications to IEEE 754 floating-point units to help researchers and scientists monitor and control errors in scientific applications as well as provide faster method for extending precision compared to modern purely software solutions. To accomplish this, support is added to the RISC-V simulation environment through gem5 architecture simulator to give the ability to identify possible elements lost during rounding and experiment with extended precision. The use of the SoftFloat arithmetic validation suite is utilized and added to gem5 for better floating-point simulations. Simulation results are presented indication good performance and the ability to monitor arbitrary precision. Results are also given on implementation in System on Chip designs using the Global Foundries cmos32soi technology along with ARM standard-cells. The results indicate an approximate 5% increase in area with less than 3% increase in energy over traditional IEEE 754 floating-point multipliers.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Multiplication has long been an important part in most computer architectures and it has usually been seen as a common case and as an design decision to include in any microarchitecture. However, the difficulty in creating hardware for multiplication because of the inherent shifting of the radix point during calculations has been a cogent reason for well-developed floating-point hardware in scientific applications. To aid common usage for floating-point in computer architectures, the IEEE standardized floating-point in 1985 and subsequently re-ratified it in 2008 with the IEEE 754.

Although IEEE 754 floating-point implementations have made tremendous progress in making computations simpler and concise, it has an inherent problem within its structure. There is a limit to the precision of a single floating-point number based on the exponent of that number. This is known as the number's dynamic range. Since the dynamic range is much larger than normal integer and fractional implementations while constrained to the same space, information can be lost when numbers are rounded to fit within the possible dynamic range of floating-point during the final steps of calculations. The IEEE 754 standard, by default, rounds floating-point numbers using round-to-nearest even or RNE and has a total of four rounding modes to help contain error. Good hardware for rounding in floating-point arithmetic is key to expanding algorithms, numerical methods, and applications that exploit techniques to control validation due to loss of precision.

In addition, correct rounding of both normal and denormal results further exacerbates the growing complexity of an IEEE 754 multiplier. Due to the importance of high precision in scientific applications, the precision must be preserved or at the very least accessible in some other way. Simply truncating denormal results to zero is unacceptable and could result in a loss of precision data that contaminates results. Consequently, having floating-point units that can handle normalized and denormalized numbers is essential, especially for scientific computing.

While most general-purpose CPU/GPU use double-precision floating point units, in deep learning, single-precision floating-point is widely used as the default format because its smaller dynamic range results in faster calculations and smaller or more parallel application-specific hardware. However, recent research shows that, in many applications, single-precision floating-point multipliers can be replaced by half precision floating-point multipliers in training deep neural networks, which have little to no impact on the network accuracy. Therefore, there is a need for a new type of multiplier that can switch between different precisions for deep learning tasks using information about the loss of precision in previous steps. Moreover, it is important that the ability to monitor error during larger chains of computations exist and be available to programs that use this hardware.

To overcome the numerical limitations of existing computer systems, several software tools and hardware modifications have been developed. Each of these tools or hardware designs use additional code or digital logic to extend the precision of floating-point arithmetic or improve the ability to monitor numerical errors. Although these methods are useful, many of these implementations impose lengthy cycle times or additional hardware that complicates their usage. This paper discusses a method that does not incur extra delays during regular use of the hardware while implementing a system called native-pair computations. This extension to the standard IEEE 754 multiplier is added to the RISC-V ISA for demonstration and testing. The ideas presented in this paper are based on the concepts presented in [3], in which the idea was originally proposed. Moreover, this paper also discusses architectural changes to support this new extension as well as the effectiveness of the new system using the `gem5` architectural simulator.

1.0.1 Software Implementations

Variable-precision software packages give the programmer explicit control over the precision of computations [4], [5]. Typically, fixed-point or floating-point hardware is used to simulate the variable-precision calculations and subroutine calls are required for each operation. A variety of applications using variable-precision software packages has been successfully developed. These include finding roots of polynomials and evaluating elementary functions [6].

To give the programmer more power and flexibility in developing numerical software, several scientific programming languages have been developed [7] [8], [9], [10]. These languages are extensions to existing programming languages that typically provide variable-precision arithmetic, interval arithmetic, and data types for vector and matrix operations. Several applications using scientific programming languages have also been successfully developed such as inverting and multiplying matrices and solving a system of nonlinear equations [11].

Another class of designs involve variable-precision processors, which are capable of performing arithmetic operations on variable-precision floating-point numbers [12] [13]. These processors extend the available precision through larger registers and memory support. Additional hardware is also available for rounding control, exception handling, and specifying the arithmetic operations.

1.0.2 Using Native-Pairs for Computing

This thesis will investigate support for native pairs in floating-point multiplier units. Instead of using dedicated functional units or coprocessors, this dissertation will focus on modifications that can be made to conventional processors to enable them to efficiently support native pairs for floating-point multiplication. It is anticipated that these modifications can also be extended for other floating-point computations, such as addition and division. This approach offers the performance benefits of dedicated hardware with only a marginal increase in area. It also lets the floating-point multiplier hardware take advantage of improvements in floating-point hardware and Very Large Scale Integration (VLSI) technology and eliminates the overhead of transferring data between the main processor and a native-pair processor or functional unit.

For this thesis, algorithms and hardware designs for a combined native-pair and floating-point multiplier will be developed. This will include the design and evaluation of functional units that can perform both extended floating-point and IEEE 754 floating-point computations, as well as datapath and control modifications needed to efficiently support native-pair data. Combining both types of operations on the same hardware will limit delay and need for complex interface hardware. The goal is to incorporate support in the design of conventional processor hardware with only a minor increase in area and little or no increase in cycle time.

These functional units will be designed, simulated, and verified using System Verilog and synthesized into a System on Chip (SoC) standard-cell implementation. Area and delay estimates for each of the designs will also be made and compared to estimates for conventional floating-point units. To investigate the performance benefits achieved by hardware support for native-pair multiplications, the gem5 toolset [14] is used to measure the performance of dedicated benchmarks that incorporate native-pairs within an IEEE 754 floating-point multiplication hardware.

1.1 Contributions of this Research

The designs that have been and will be developed will help to improve upon the numerical accuracy and reliability of computer systems. This has a potential of impacting a large number of fields in engineering and applied sciences that depend on computer simulation and modeling. The following are potential contributions from this research:

1. Hardware designs for interval arithmetic with minimal impact on area and delay.
2. Efficient algorithms and designs for extending precision within normal IEEE 754 floating-point multiplication computations.
3. Architecture support for combined native-pair and IEEE 754 floating-point multiplications targeted at Reduced-Instruction Set Computer (RISC) architectures.
4. A better understanding of the design of instruction set and hardware designs for computer systems that are targeted toward scientific applications.

1.2 Organization

The organization of this thesis is as follows: Chapter II will cover IEEE 754 floating-point and past architectures that implemented native pairs and their pitfalls. Chapter III explains this paper's implementation of native pairs, how it solves previous designs' problems, and how it can be used beyond just monitoring error. Chapter IV showcases SPEC benchmarks and their functional unit performance metrics in a `gem5` execution environment using the SoftFloat library extension. Finally, Chapter V presents the conclusion to the work in addition to possible future research on the topic.

CHAPTER II

BACKGROUND

The expansion of hardware to allow an increased amount of precision or more accurate results is important for scientific computing. Although IEEE 754 floating-point arithmetic is powerful, it can consume a large amount of space in a design as well as have an impact on the cycle time and overall performance of the system. Over the years there has been many attempts to leverage hardware against simplicity while continuing to maintain good performance.

One class of designs involves the computation of accurate dot products. Accurate dot product coprocessors produce dot products that are mathematically exact, but have a single rounding at the end. The ability to allow floating-point numbers to be accumulated without roundoff error is accomplished using a long fixed-point accumulator. A long fixed-point accumulator (LA) that ensures exact accumulation requires

$$L = g + 2 \cdot E_{max} + 2 \cdot |E_{min}| + 2 \cdot l + 1$$

digits, where the input floating-point format in terms of the base numbers have a mantissa of length l and exponent range from E_{min} to E_{max}. The g additional bits, called guard bits, are used for catching intermediate overflows. After the accumulation, the exact dot product in the LA is rounded once to the desired floating-point format using one of the four rounding modes specified by the IEEE 754 standard [15].

Dot product coprocessors use memory to load and store the accumulator, a barrel shifter to find the correct point to add new numbers to the accumulator, and an adder or subtractor. Designs are presented in [16], [17], and an overview of accurate vector

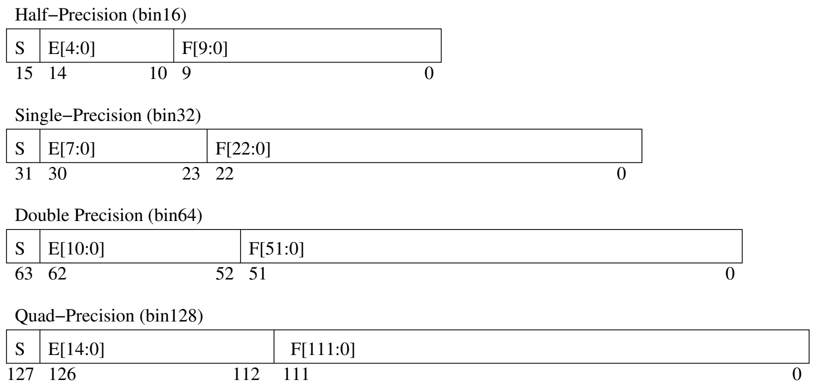

Figure 2.1: Data formats for the IEEE 754-2008 floating-point

arithmetic units is given in [18]. In addition, [17] presents carry-skip logic to determine if a carry-chain can be bypassed in the accumulator, based on a solution previously implemented in software [19]. A two-bit wide register is attached to each accumulator word, where one bit indicates all digits of the corresponding LA word are zero, and the other bit indicates all digits are $(b-1)$. The carry skips over word boundaries is based on this two-bit flag. Unfortunately, LA and other validated-arithmetic implementations require additional software support and can easily complicate hardware arithmetic units.

The IEEE 754 floating-point standard, originally ratified in 1985 [15] and later amended in 2008 [20], defines the floating-point format that consists of three parts: sign (S), exponent (E), and mantissa or significand (M). Figure 2.1 shows four IEEE 754 formats including half-precision, single-precision, double-precision and quadruple-precision formats. IEEE 754 floating-point arithmetic provides a modest increase in hardware while providing user-accessible support for increased precision that cannot be easily handled through integer arithmetic. Floating-point support within the RISC-V architecture is handled through the “F”, “D”, and “Q” standard extension for single, double, and quadruple precision, respectively.

Figure 2.2 shows a block diagram detailing the overall architecture. The design consists of several stages: unpack (hidden bit and other exception and bit testing),

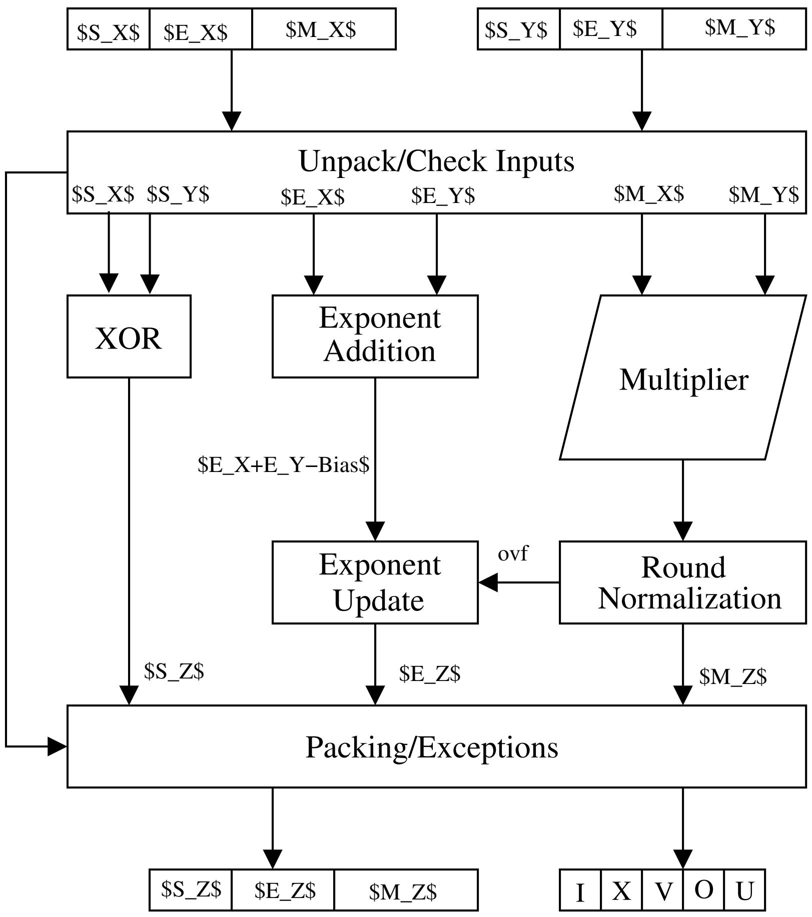

Figure 2.2: Block diagram of IEEE 754 multiplier architecture

sign, exponent and mantissa logic blocks and final result packing. As per the IEEE 754 standard, five flags are produced: Infinite or Divide by 0 (I), Inexact (X), Invalid (V), Overflow (O) and Underflow (U). Some flags, such as Divide by 0, are not appropriate for floating-point multiplication as it is not possible.

As stated previously, although IEEE 754 arithmetic is now standardized and commonplace in most general-purpose and application-specific processors, it does suffer from loss of information due to rounding the final result to its IEEE 754 representation. This error, although small, can possibly compromise applications where error in precision is a critical element in its use (e.g., conversion between integers and IEEE 754 arithmetic). Therefore, the need for architectures to be able to *analyze* error during use is important for high-performance computing and their applications.

A more pragmatic solution to this problem is utilizing something called native-pair arithmetic [1]. A pair of native floating-point numbers are used to represent a base result and a residual term which is used to increase accuracy by storing normally-discarded precision. The original idea [1] adds a few simple microarchitectural features so that acceptable accuracy can be obtained with a relatively little performance penalty. To reduce the cost of native-pair arithmetic, a residual register is used to hold information that would normally have been discarded after each floating-point computation.

The main idea here is to balance hardware and software by providing a sequence of numbers that can be used for arbitrary precision [21]. In theory, this could allow a group of several numbers to approximately double the amount of precision for a computation without the inclusion of additional hardware [22]. As pointed out in [1], one issue with native-pair arithmetic, or sometimes called *double-double* when used with IEEE 754 double-precision floating-point numbers, is that it can take up to ten or more native operations for each native pair operation.

To accomplish this task, a residual register [1] is suggested that takes in the discarded values saved by the IEEE 754 floating-point units (FPUs). This residual register stores unnormalized results, but utilizes the same IEEE 754 floating-point hardware that exists for computing the residual register. After computation, the new instruction `MOVRR` that has been added to the Instruction Set Architecture (ISA) is used to handle moving the residual register's value into the register file. The overall architecture looks like the architecture in Figure 2.3. The residual register is a floating-point register with a sign bit, n_e exponent bits, $n_m + 2$ mantissa bits, and a complement flag bit, where n_e and n_m are the number of exponent and mantissa bits in a native floating-point number, respectively, not including the leading one bit in the mantissa implied by the IEEE 754 format [1]. Programs that do not use the residual register get the usual result defined by the IEEE 754 standard. Most

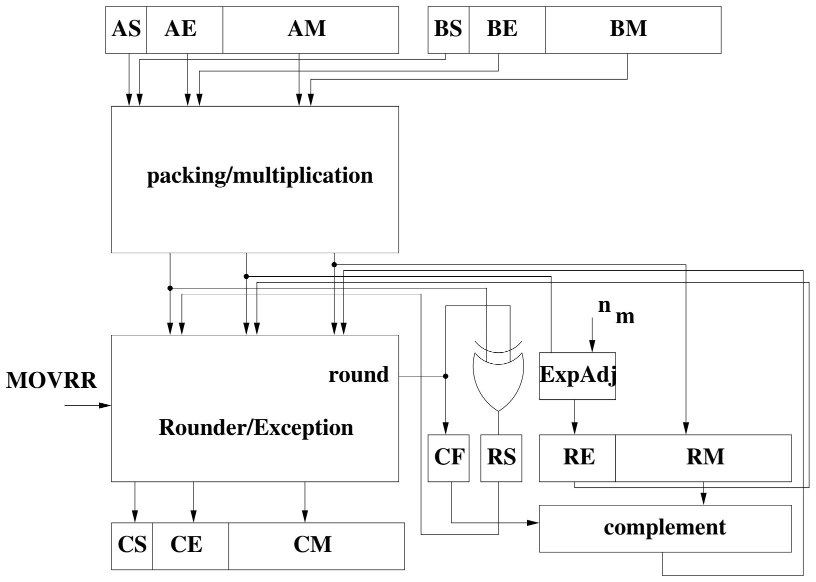

Figure 2.3: Previously Proposed Architecture for Residual Register in IEEE 754 Floating-Point Multiplier [1]

importantly, results stored in the residual register (prefixed by `R`) can be used to speed up extended-precision floating-point algorithms by replacing sequences of instructions that compute equivalent results with a single residual register access [1].

This architectural design allows a good compromise between the complexity and rest of the system's architecture needs. The `MOVRR reg, K` instruction in the ISA allows the compiler to easily control scheduling and possibly remove any hazards when multiplier instructions produce residual results, especially in out-of-order systems [1]. Although the design in Figure 2.3 shows the change for IEEE 754 multiplication, the original idea in [1] can be applied to other IEEE 754 floating-point operations, as well.

The difficulty in rounding is due to the IEEE 754 standard's format for the mantissa being in the correct range. This typically means that logic has to check whether the 106-bit product (i.e., $P[105:0]$) of the multiplication for the correct values of l, g, and t. This means that if $v = 0$ (no overflow), $l = P[52]$, $g = P[51]$ and t is the logical OR of $P[50:0]$, however, if $v = 1$ (overflow), $l = P[53]$, $g = P[52]$ and t is the logical OR of $P[51:0]$. The rounding bit r is then added to the least-significant bit (LSB) (which is $P[52]$ if there is no overflow and is $P[53]$ if overflow) by a 54-bit carry-propagate adder (CPA).

Multiplication is basically adding the multiplicand multiple times based on values of the multiplier [23]. To speed this process up, parallel multipliers, typically found in IEEE 754 multipliers, use a carry-save format so that it can avoid the slow 106-bit CPA until later in the process [24]. This carry-save format allows the product to be computed optimally by paralleling the addition of each partial product. Consequently, the mantissa multiplication within IEEE 754 multipliers generates the partial products and then reduces it to a carry-save format that includes 106-bit carry $C[105:0]$ and a 106-bit sum $S[105:0]$ vectors.

CHAPTER III

NATIVE PAIR IMPLEMENTATION

Due to the multiplier presenting its product in carry-save format to the rounder, it is difficult to determine if there is an overflow (i.e., $P >= 2$) [25]. In order to help optimize the hardware, parallel additions are performed and additional logic is utilized to determine which additions are utilized for the final product. These parallel additions are combined together to form one adder, typically called a compound adder (CA). Compound adders take advantage of utilizing redundant hardware and its use is critical in optimizing hardware for any implementation [24]. Normally, compound adders use the same hardware except for critical components, such as the carry-chain logic [25].

Round to Nearest (RN) is arguably the most complicated mode compared to Round to Zero (RZ) and Round to Infinity (RI) modes. The method within [25] smartly designs for round-to-nearest/up (RNU) mode (roundTiesToAway mode in IEEE 754 standard) and then modifies the design to produce RN mode. The RNU mode utilizes RN mode except in the case of a tie ($x.rem = 0.5$) where the RNU mode always rounds up. In terms of implementation, RNU can be implemented by simply adding a 1 to the guard bit (g) position. This introduced error, although small, can build over time and eventually cause problems [26].

Native-pair computations can be utilized to essentially build on top of current operations to create multi-precision computations [27, 22]. Essentially, for multiplication this is done as a straightforward multiplication followed by accumulation of the results. Luckily, this process does not have problems associated with catastrophic cancellation or the subtracting of two closely related values [27]. Accumulation can be sped up by having architectures that have fused-multiply and add (FMA) or sometimes called multiply and accumulate (MAC) units, however, most common ISAs do not have this instruction. For multiplication, the most important operation is guaranteeing that no significant digits are lost when the product of two components is computed with its limited precision [21].

3.1 Explanation of Native Pairs

As specified in [22], using multiple components and splitting their computations and accumulating them later is called native-pair floating-point computations in this paper, similar to [1]. It is argued in this paper that simpler architectural changes are needed that do not strangle other operations or more specifically that *make the common case fast*. Although it is conceivable to perform this native-pair operation for any floating-point computation, this work makes the argument that this architecture modification can be done if a user wants to examine more information about a given floating-point computation and uses multiplication as the basis. Granted, this operation, would consume more execution time than a normal non-native-pair floating-point program, however, the ability to save the extra bits of precision by the floating-point unit can be significant in power to a user who might be concerned with very small or large numbers or, worse, possible loss in precision. Therefore, using the native-pair computations, as suggested by [1], is a good trade-off between complexity and simplicity.

What makes this modification challenging is the post-normalization step or the rounded product needs to be normalized (divided by 2) for the mantissa domain $[1, 2)$ by a right shift if it is equal to or larger than 2. The current implementation in [1] does not use current architectures that well known for IEEE 754 floating-point architectures [25, 28, 29]. This research has shown good architectures to optimize

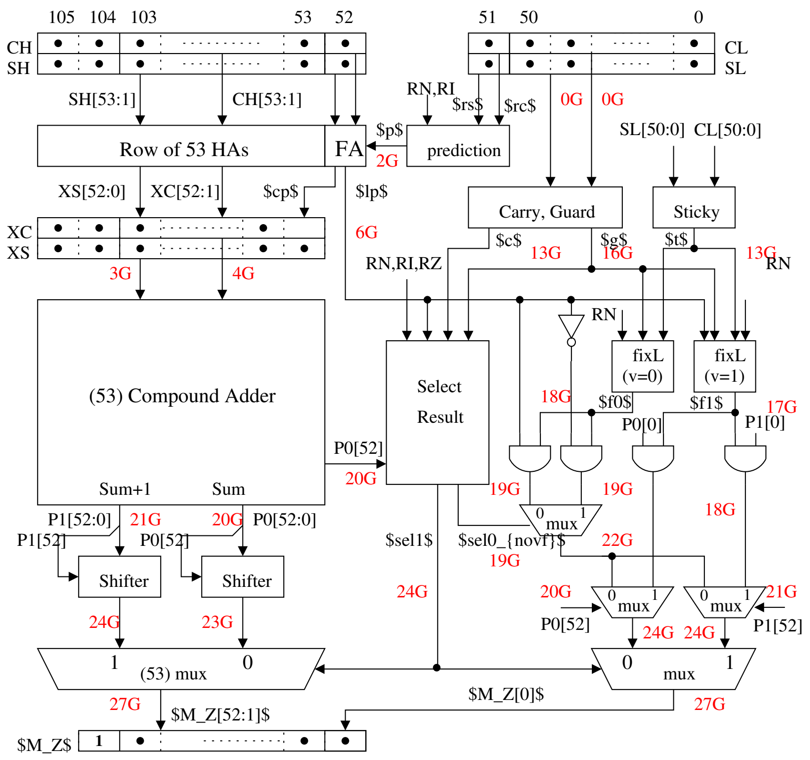

Figure 3.1: Rounding architecture for all IEEE 754 rounding modes (Adapted from [2])

one of the main delay issues within IEEE 754 multiplication, the rounder. Recent research [30, 2] has given further optimizations into this critical part by analyzing each design. This optimized rounder can be seen in Figure 3.1.

Figure 3.1 shows an optimized rounder unit that starts with inputs from the 106-bit carry-save output (i.e., `CL[105:0]` and `SL[105:0]`) from the multiplication unit. The upper 54 most-significant bits (MSBs) from or `SH`, `CH` and the 52 least-significant bits (LSBs) for `SL`, `CL` ($PL = SL + CL$), respectively, are separated to speed up the critical path within this unit. The left-hand portion of the block in Figure 3.1 utilizes a row of 53 HAs to add `SH` and `CH` (except the LSBs) and one FA to add the prediction bit `p` and two LSBs of `SH`, `CH`. The sum bit `lp` is used to compute the correct LSB of final product on the right while the carry bit `cp` is added into the LSB of the carry vector `XC` on the left. A 53 bit compound adder is then used to pre-compute two

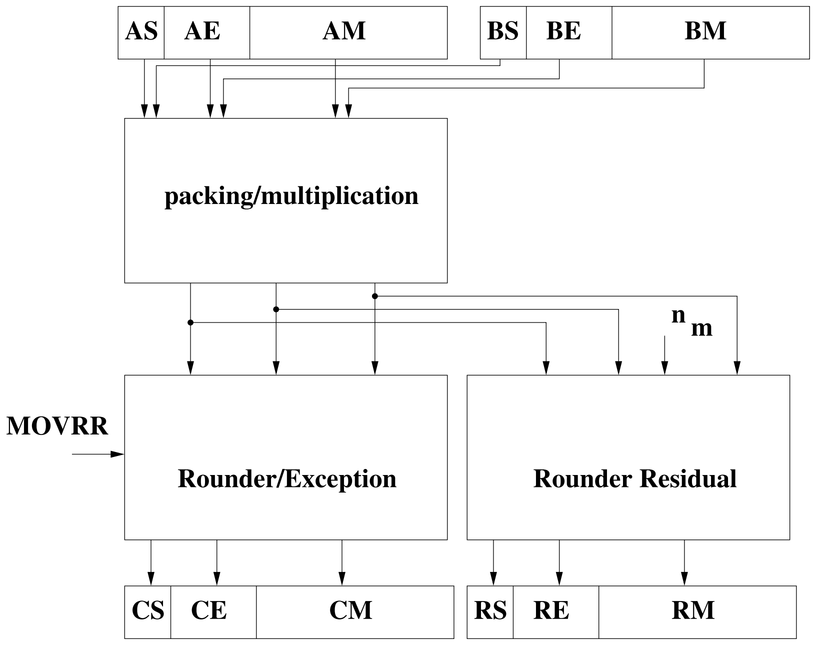

Figure 3.2: Proposed Architecture for Residual Register in IEEE 754 Floating-Point Multiplier

possible outputs `P0`, `P1` [2]. Both `P0` and `P1` are normalized before the final selection logic. On the right side of Figure 3.1, the carry `c`, guard `g`, and sticky `t` bits are computed based on `SL` and `CL` bits [2]. Based on the last bit of `lp` and `c`, `g`, `t` bits and the overflow bit $v_0 = P0[52]$, the **Select Result** module generates `sel1` and `sel0` signals to select the correct output from the CA based on the correct value of `INC`.

In Figure 3.1 annotated linear-delay numbers to give a theoretical idea of the delay encountered by this unit. Linear-delay analysis is a useful technique to analyze Boolean logic [24]. Typically, a set amount of delay is universally set for each gate within a module and each implementation uses only those gates in the library to perform a comparison. This way, a design can be compared individually and without bias. In this figure, delays are annotated with the letter `G` to signify "gate delays" as an arbitrary delay unit. As seen in Figure 3.1, the normalization signal, `sel1` set by the **Select Result** unit, consumes 24G delays. This signifies that the logic in Figure 2.3 requires a significant amount of delay before the residual register can be computed. As seen in Figure 3.1, once the `sel1` signal is asserted or de-asserted, it would require an additional 27G to be re-introduced through the rounder unit before even producing an answer in Figure 2.3. Unfortunately, this would be prohibitive for most high-performance computing applications and a better solution is needed.

One potential solution is to replicate the rounder unit within the IEEE 754 multiplication unit. The secondary rounder is utilized to separately compute the residual value. This architecture has the advantage in not having to wait for the normalization signal. The `MOVRR` control signal is still needed to signal the final result to select the residual register through a multiplexor (not shown in Figure 3.2) as an output instead of a normalized IEEE 754 floating-point result. Theoretically, this unit could also supply this information as an additional output, however, this would require an infrastructural change within the microarchitecture to handle the additional outputs from IEEE 754 FPUs.

3.2 Example Use of Residual Register and MOVRR Instruction

To demonstrate how this implementation works, an example is given for native-pair multiplication based on the work in [21]. C++ programs were written to prove that the production of native-pair computations can provide precision much larger that is needed if this extra information is available to a user. The source code for both of these programs are in appendix A for single precision and appendix B for double precision.

After a floating-point multiplication instruction is completed within a system using a residual register, the value within that register can be accessed and moved to a general purpose floating-point register with the `MOVRR` instruction, similar to the instruction originally proposed in [1]. For example, given two single precision floating-point values `x` and `y`, the resulting product of the two along with using `MOVRR` to recover the lost precision finishes with two registers that contain the full product.

```
x = 4.00000095367431640625
  = 0x4080_0002
y = 2.0000002384185791015625
  = 0x4000_0001


z = x * y
z[1] = 8.0000028610229492 1875
     = 0x4100_0003
z[2] = 0.00000000000022737367544323205947875976 5625
     = 0x2a80_0000
```

The result of multiplying the same two values `x` and `y` but with double precision gets the same product but with the entire answer in a single register.

```
z = (double) x * y
z = 8.000002861023176592425443232059478759765625
  = 0x4020_0000_6000_0080
```

The C++ code used to generate these outputs can be found in appendix A.

This can be extended further with a double precision multiplier and multiplicand, demonstrating the ability to go beyond what most hardware IEEE 754 multipliers naively.

```
x = 4.000000000000001776356839400250464677810668945 3125
  = 0x4010_0000_0000_0002
y = 2.0000000000000004440892098500626161694526672363 28125
  = 0x4000_0000_0000_0001
```

```
z = x * y

z[1] = 8.000000000000005329070518200751394033432006835937 5

     = 0x4020_0000_0000_0003

z[2] = 0.00000000000000000000000000000078886090522101180541 ...

          172856528278622967320643510902300477027893066406 25

     = 0x39b0_0000_0000_0000
```

The result when using a system that supports quad precision or using software libraries to make up for the lack of quad precision support again results in the same answer to contained within a single register instead of being split into a native pair.

```
z = (quad) x * y

z = 8.000000000000005329070518200752182894337227847742911 72 ...

      85652827862296732064351090230047702789306640625

  = 0x4002_0000_0000_0000_3000_0000_0000_0200
```

The C++ code used to generate these outputs can be found in appendix B.

The solution using the residual register and MOVRR contains the same numeric value, but the representation is split between two double precision floating-point registers and thus can be used in systems that do not have support for IEEE 754 quadruple precision at the hardware level. Even though RISC-V has quadruple-precision support, this technique can be utilized for larger precisions, if needed. Existing IEEE 754 floating-point implementations remove or erase this extra information within most floating-point units (FPUs), thus, this modification provides good support for those pursuing areas of accuracy within a given amount of precision.

As documented in [26], there are many numerical packages that can examine extra information about a specific computation. In addition, existing GNU repositories utilize libraries for possible multiple-precision floating-point computation (e.g., GNU MPFR). On the other hand, all of these software tools consume large amounts of execution time and do not utilize hardware to help alleviate execution times. It is suggested within this work that utilizing more information within FPUs can help optimize and examine numerical issues that exist with computer arithmetic computations.

CHAPTER IV

BENCHMARKING

To demonstrate the effects the residual register has on runtime performance, a modified version of the RISC-V instruction set architecture that contained the new register was used in the `gem5` simulator. This particular RISC-V setup used the RV64I base as well as the G subset of extensions. The simulator is set up in system call emulation mode, allowing for benchmarks and example programs to be run without setting up an operating system. Typical setup values utilized within `gem5` are shown in Table 4.1. For benchmarking, a series of SPEC benchmarks that emphasized floating-point instructions are used to gauge system performance with the new register in place. A total of six SPEC benchmarks are used from both the 2006 and the 2017 edition of SPEC CPU benchmarks, as shown in Table 4.2 and Table 4.3.

The system model in `gem5` uses an out-of-order CPU with one processor. Each benchmark was run single-threaded on their own instance and had 64kB of L1 instruction cache and 32Kb of L1 data cache. Each instance was given 16GB of simulated memory simulating DDR4 2400MHz timing and performance. Only one memory channel was used for these particular benchmarks. In order to simulate floating-point performance within the `gem5` simulation, proven floating-point software routines are added to the `gem5` simulator. These routines, called `SoftFloat` [31], are routines utilized for testing floating-point implementations as well as testing them against hardware. `SoftFloat` is efficiently written in C and can be integrated within the `gem5` simulator. The `SoftFloat` routines are based on routines originally devised within the PARANOIA program written by W. Kahan [32]. An additional instruc-

CPU Architecture	RISC-V
CPU Type	DerivO3CPU
L1d Cache Size	64kB
L1i Cache Size	32kB
Memory Type	DDR4_2400_8x8
Memory Size	16GB
Memory Channels	1

Table 4.1: gem5 Simulation Specifications

tion is also integrated, `MOVRR`, to allow extra information to be presented to a user, if needed. Although the SPEC CPU benchmarks do not employ this extra instruction, the idea is that this capability can be employed to examine specific precision. Simulations through `gem5` indicate no foreseeable negative consequence to a simulation other than adding an additional instruction through the Instruction Set Architecture (ISA).

As seen by the results of the SPEC06 benchmarks in Table 4.2 and the SPEC17 benchmarks in Table 4.3, demanding floating-point computations can be a significant amount of a program's execution time. Moreover, any additional program that uses accurate, self-validating arithmetic potentially could consume much more execution time as it utilizes libraries that are typically slower and have high amounts of overhead. For example, specific software packages that employ computations, such as interval arithmetic, typically use directed roundings or round-to-positive and negative infinity. These directed roundings, although part of the IEEE 754 standard [15, 20], typically are controlled by the Floating-Point Status and Control Register within the RISC-V architecture. And, if any changes are required during a complicated floating-point pipeline, many architectures flush the pipeline to avoid issues with complicated changes in the rounding mode.

SPEC06 Benchmark	444.namd	470.lbm	508.namd_r	519.lbm_r
Runtime Information				
Simulated Seconds	17.55132	10.259652	10.034215	1,482.954635
Real Seconds Elapsed	101,470.81	28,745.83	67,628.68	4,778,013.15
# of Simulated Cycles	35,102,640,085	20,519,304,925	20,068,429,170	2,965,909,270,356
Function Frequency				
Total Function Calls	47,413,825,438	6,610,717,022	34,198,662,665	1,595,256,157,701
FloatADD	5,917,003,819	2,273,240,080	3,743,658,184	541,500,665,712
FloatMULT	4,414,971,460	1,273,446,080	3,294,084,157	326,750,716,512
% of Runtime				
FloatADD	12.48%	34.39%	10.95%	33.94%
FloatMULT	9.31%	19.26%	9.63%	20.48%

Table 4.2: Results of SPEC06 RISC-V gem5 simulations

The modifications provided in this work do not incur any extra architectural changes other than more area within the FPU. Synthesis was performed on the two IEEE 754 multiplier designs, one with `MOVRR` support, and one traditional. Results were obtained with the cmos32soi 32nm technology using ARM standard-cells and synthesis was performed using topographical synthesis. Topographical synthesis, provided by Synopsys® DC™ (DC) ensures synthesis that accurately predicts timing, area and power by including information from the standard-cell layouts and underlying interconnect. Results indicate a 6.48% (17.755 mm^2 traditional vs. 18.907 mm^2 with `MOVRR`) increase in area with no delay addition. The energy consumption also increases due to more area utilized for the architecture modification. The average power estimation is achieved by running the simulation with over $46,464$ random test vectors generated by TestFloat [31] utilizing an annotated Value Change Dump (VCD) and subsequently converted to a Switching Active Interchange Format (SAIF) for analysis through DC topographical. Results indicate a 2.32% increase in energy (30.59 mW traditional vs 31.30 mW with `MOVRR`). This increase is very small and in

SPEC17 Benchmark	619.lbm_s	644.nab_s
Runtime Information		
Simulated Seconds	75.938858	2.635289
Real Seconds Elapsed	220,672.45	17,830.05
# of Simulated Cycles	151,877,716,470	5,270,577,837
Function Frequency		
Total Function Calls	5,225,632,9490	8,446,078,443
FloatADD	15,594,087,856	990,094,464
FloatMULT	9,024,897,856	1,192,676,971
% of Runtime		
FloatADD	29.84%	11.72%
FloatMULT	17.27%	14.12%

Table 4.3: Results of SPEC17 RISC-V gem5 simulations

the situation that users take advantage of the residual register, power can be saved in other ways such as not using the high precision functionality of a given architecture or using software to make up extra precision through multiple successive operations.

CHAPTER V

CONCLUSION AND FUTURE RESEARCH

This paper demonstrates an add-on to IEEE 754 floating-point that adds functionality for capturing and measuring error in floating-point operations during normal use of those operations without incurring any additional delay. Those captured error values are then made available for use through a newly-added instruction `movrr` that only requires the delay delay a single register read and write. Having access to this additional error information as presented in this paper through an IEEE 754 multiplier opens many possibilities in both software extensions and further hardware expansion. Not only is it possible to verify the results of floating-point operations that take place and make sure rounding during those operations has not compromised the overall precision of the final result, these error measurements can be used in software libraries to increase the precision even when the native hardware including this error-managing architectural change does not have support for higher precision. Verification of this change at the hardware level through Verilog HDL shows only slight increases in power consumption and logic area with no change in delay, meaning the proposed architectural modification does not impose any new delays or decrease the performance of already-existent hardware. Further testing was performed to show the usefulness of such an operation through C++ programs that demonstrate the potential for accelerated native pair floating-point operations using the residual register and the `movrr` instruction.

In addition to the ideas presented on the residual register, changes to the RISC-V `gem5` architecture simulator to support the new residual register and `movrr` instruction. This required changes to the method that `gem5` uses to handle floating-point operations which up until now involved using the hardware's native floating-point operations for simulation results. By adding the `SoftFloat` library to `gem5`, which supports full IEEE 754 operations in a manner that makes their internals visible for further use, floating-point operations become much easier to modify and expand - a necessity for this particular modification with the residual register. The RISC-V toolchain used to compile programs that run on processors that use the RISC-V architecture was also modified to support the new `movrr` instruction at the assembly level, allowing the use of inline assembly calls in C and C++ programs in order to use the instruction within a standard program.

Finally, the RISC-V `gem5` simulator with support for the `SoftFloat` library was used to benchmark many of the SPEC benchmarks commonly used in computer architecture research to gauge performance of a system and compare that performance with that of other systems. This shows off the huge potential that many of these benchmarks have when it comes to improving floating-point performance that could be done using the residual register hardware and software additions presented here. Further work can be done by digging deeper into the benchmarks to measure the individual error in each floating point operation and accumulate it over the course of the benchmarks to see where unacceptable levels of precision are lost due to rounding and potentially fix those errors using this proposed hardware.

Future work on this subject will entail adding the residual register functionality to the floating-point add and divide operations to give complete coverage of main IEEE 754 floating-point functionality. With all major operations fitted with this architectural modification, its usefulness and flexibility drastically increase for almost any workload. Also, more research into the software solutions that can take advantage of this hardware change, such as real-time error feedback and on-the-fly precision adjustments, along with their implementations in a simulator like `gem5` will be helpful in further use cases for this modification. The careful design of software libraries are where much of this architecture's abilities can be passed on to programmers who may not even be aware of what the modification is doing at the low level but can benefit from what the architecture provides.

REFERENCES

[1] W. R. Dieter, A. Kaveti, and H. G. Dietz, “Low-cost microarchitectural support for improved floating-point accuracy,” *IEEE Computer Architecture Letters*, vol. 6, pp. 13–16, Jan 2007.

[2] T. D. Nguyen, S. R. Thompson, and J. E. Stine, “Architectural improvements in IEEE-compliant floating-point multiplication,” *submitted to IEEE Transactions on Computers*, 2018.

[3] A. Underwood and J. E. Stine, “IEEE floating-point extension for containing error in the RISC-V architecture,” in *Proceedings of the Third Workshop on Computer Architecture Research with RISC-V*, CARRV’19, (New York, NY, USA), ACM, 2019.

[4] R. P. Brent, “A FORTRAN Multiprecision Arithmetic Package,” *ACM Transactions on Mathematical Software*, vol. 4, pp. 57–70, 1978.

[5] F. C. Motteler, “Arbitrary Precision Floating-point Arithmetic,” *Dr. Dobb’s Journal*, pp. 28–34, September 1993.

[6] R. P. Brent, “Multiple-Precision Zero Finding Methods and the Complexity of Elementary Function Evaluation,” in *Analytic Computational Complexity* (J. F. Traub, ed.), pp. 151–176, Academic Press, Inc., 1976.

[7] J. H. Bleher, A. E. Roeder, and S. Rump, “ACRITH: High Accuracy Arithmetic - An Advanced Tool for Numerical Computation,” in *Proceedings of the 7th Symposium on Computer Arithmetic*, pp. 318–321, 1985.

[8] J. H. Bleher, S. M. Rump, U. Kulisch, and J. W. von Gudenberg, "FORTRAN-SC: A Study of a FORTRAN Extension for Engineering Scientific Computation with Access to ARITH," *Computing*, vol. 39, pp. 93–110, 1987.

[9] C. Falco-Korn, S. Koenig, and S. Gutzwiller, "MODULA-SC: A Precompiler to Modula-2," in *Contributions to Computer Arithmetic and Self-Validating Numerical Methods* (C. Ullrich, ed.), pp. 371–384, J.C. Baltzer, 1991.

[10] J. Wolff von Gudenberg, "PASCAL-SC: A PASCAL Extension for Scientific Computation," in *Proceedings of the 10th IMACS World Congress on System Simulation and Scientific Computation*, pp. 402–408, 1982.

[11] G. F. Corliss, "Industrial Applications of Interval Techniques," in *Computer Arithmetic and Self-Validating Numerical Methods* (C. Ullrich, ed.), pp. 91–113, Academic Press, 1990.

[12] T. E. Hull, M. S. Cohen, and C. B. Hull, "Specification for a Variable-Precision Arithmetic Coprocessor," in *Proceedings of the 10th Symposium on Computer Arithmetic*, pp. 127–131, 1991.

[13] M. S. Cohen, T. E. Hull, and V. C. Hamacher, "CADAC: A Controlled-Precision Decimal Arithmetic Unit," *IEEE Transactions on Computers*, vol. C-32, pp. 370–377, 1983.

[14] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[15] "IEEE standard for binary floating-point arithmetic," *ANSI/IEEE Std 754-1985*, pp. 1–14, 1985.

[16] P. R. Capello and W. L. Miranker, "Systolic super summation," *IEEE Transactions on Computers*, vol. 37, pp. 657–677, June 1988.

[17] A. Knofel, "Fast hardware units for the computation of accurate dot products," in *[1991] Proceedings 10th IEEE Symposium on Computer Arithmetic*, pp. 70–74, June 1991.

[18] G. Bohlender, "What Do We Need Beyond IEEE Arithmetic?," in *Computer Arithmetic and Self-Validating Numerical Methods* (C. Ullrich, ed.), pp. 1–32, Academic Press, 1990.

[19] M. Muller, C. Rub, and W. Rulling, "Exact accumulation of floating-point numbers," in *[1991] Proceedings 10th IEEE Symposium on Computer Arithmetic*, pp. 64–69, June 1991.

[20] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[21] D. M. Priest, "Algorithms for arbitrary precision floating point arithmetic," in *[1991] Proceedings 10th IEEE Symposium on Computer Arithmetic*, pp. 132–143, June 1991.

[22] T. Dekker, "A Floating-Point Technique for Extending the Available Precision," *Numerische Mathematik*, vol. 18, pp. 224–242, 1971.

[23] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware Software Interface ARM Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2016.

[24] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2003.

[25] M. R. Santoro, G. Bewick, and M. A. Horowitz, "Rounding algorithms for IEEE multipliers," in *Proceedings of 9th Symposium on Computer Arithmetic*, pp. 176–183, Sep 1989.

[26] M. J. Schulte and E. E. Swartzlander, Jr., *Software and Hardware Techniques for Accurate, Self-Validating Arithmetic.* Kluwer Academic Publishers, 1996.

[27] Y. Hida, X. S. Li, and D. H. Bailey, "Algorithms for quad-double precision floating point arithmetic," in *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*, pp. 155–162, June 2001.

[28] G. Even and P.-M. Seidel, "A comparison of three rounding algorithms for IEEE floating-point multiplication," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 638–650, 2000.

[29] N. T. Quach, N. Takagi, and M. J. Flynn, "Systematic IEEE rounding method for high-speed floating-point multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 5, pp. 511–521, 2004.

[30] T. D. Nguyen, S. Bui, and J. E. Stine, "Clarifications and optimizations on rounding for IEEE-compliant floating-point multiplication," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 1–8, July 2018.

[31] J. Hauser, "The SoftFloat and TestFloat Validation Suite for Binary Floating-Point Arithmetic," tech. rep., University of California, Berkeley, 2018. Available at http://www.jhauser.us/arithmetic/TestFloat.html.

[32] W. Kahan, "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic," tech. rep., University of California, Berkeley, 1996. Available at http://www.cs.berkeley.edu/˜ wkahan.

[33] M. Daumas and D. W. Matula, "Validated roundings of dot products by sticky accumulation," *IEEE Transactions on Computers*, vol. 46, pp. 623–629, May 1997.

[34] J. Garland and D. Gregg, "Low complexity multiply accumulate unit for weight-sharing convolutional neural networks," *IEEE Computer Architecture Letters*, vol. 16, pp. 132–135, July 2017.

[35] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," *SIGPLAN Not.*, vol. 50, pp. 1–11, June 2015.

[36] G. L. Steele, Jr. and J. L. White, "How to print floating-point numbers accurately," *SIGPLAN Not.*, vol. 25, pp. 112–126, June 1990.

[37] "IEEE standard for interval arithmetic," *IEEE Std 1788-2015*, pp. 1–97, June 2015.

[38] N. J. Higham, *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, second ed., 2002.

[39] N. Quach, N. Takagi, and M. Flynn, *On fast IEEE rounding.* Computer Systems Laboratory, Stanford University, 1991.

[40] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, pp. 21–29, Mar 2018.

[41] M. Courbariaux, Y. Bengio, and J. David, "Low precision arithmetic for deep learning," *CoRR*, vol. abs/1412.7024, 2014.

[42] G. Gerwig, H. Wetter, E. Schwarz, J. Haess, C. Krygowski, B. Fleischer, and M. Kroener, "The IBM eserver $z990$ floating-point unit," *IBM Journal of Research and Development*, vol. 48, pp. 311–322, May 2004.

[43] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous CPU-GPU simulator," *IEEE Computer Architecture Letters*, vol. 14, pp. 34–36, Jan 2015.

[44] Cohen, Hull, and Hamacher, "CADAC: A controlled-precision decimal arithmetic unit," *IEEE Transactions on Computers*, vol. C-32, pp. 370–377, April 1983.

[45] T. E. Hull, M. S. Cohen, and C. B. Hall, "Specifications for a variable-precision arithmetic coprocessor," in *[1991] Proceedings 10th IEEE Symposium on Computer Arithmetic*, pp. 127–131, June 1991.

[46] T. E. Hull and M. S. Cohen, "Toward an ideal computer arithmetic," in *1987 IEEE 8th Symposium on Computer Arithmetic (ARITH)*, pp. 131–138, May 1987.

[47] A. Peleg and U. Weiser, "MMX technology extension to the intel architecture," *IEEE Micro*, vol. 16, pp. 42–50, Aug 1996.

[48] S. Oberman, G. Favor, and F. Weber, "AMD 3DNow! technology: architecture and implementations," *IEEE Micro*, vol. 19, pp. 37–48, March 1999.

[49] T. M. Carter, "Cascade: hardware for high/variable precision arithmetic," in *Proceedings of 9th Symposium on Computer Arithmetic*, pp. 184–191, Sep. 1989.

[50] E. E. Swartzlander, Jr., "Merged arithmetic," *IEEE Transactions on Computers*, vol. C-29, pp. 946–950, Oct 1980.

APPENDICES

APPENDIX A: SINGLE PRECISION C++ CODE EXAMPLE

```
/*
 * Algorithms for each function in this program were taken
 * from the paper: "Algorithms for Arbitrary Precision
 * Floating Point Arithmetic" by Douglas M. Priest.
 *
 * It is important to note that the original algorithms
 * are defined using a 1-base indexing system.  For this
 * implementation, they have been adjusted to use 0-base
 * indexing to match how C++ arrays work.
 *
 * Author: Alex Underwood
 *         alexander.underwood@okstate.edu
 */

// Remove this define to simply see the outputs and no
   intermediate steps.
//#define LIST_STEPS

#include <iostream>
#include <cstring>
#include <cmath>
#include <vector>
#include <algorithm>
#include <tuple>

/*
 * procedure sum_err()
 *
 * Calculates the sum of 2 floating point numbers and
 * returns the bits that exceed the size of the mantissa
 * as a second, normalized floating-point number.
 */

std::tuple<float, float> sum_err(float a, float b)
{
  if(std::abs(a) < std::abs(b))
  {
  float temp = a;
  a = b;
  b = temp;
  }

  float c = a + b;
  float e = c - a;
  float g = c - e;
  float h = g - a;
  float f = b - h;
  float d = f - e;

  if((d + e) != f)
  {
  c = a;
  d = b;
  }

  return {c, d};
}

/*
 * procedure add()
 *
 * Adds 2 floating-point numbers in the form of 2 vectors
 * each containing multiple floating-point numbers.  The
 * result is a new vector containing multiple floating-
 * point numbers representing the sum of the inputs.
 */

std::vector<float> add(std::vector<float> &x, std::vector<
   float> &y)
{
  int i = x.size() - 1;
  int j = y.size() - 1;
  std::vector<float> e(x.size() + y.size());

  if(std::abs(x[i]) < std::abs(y[j]))
  {
    while((i > 0) && (std::abs(x[i - 1]) <= std::abs(y[j])
       ))
    {
      e[i + j + 1] = x[i];
      i = i - 1;
    }
  }
  else if(std::abs(x[i]) > std::abs(y[j]))
  {
    while((j > 0) && (std::abs(y[j - 1]) <= std::abs(x[i])
       ))
    {
      e[i + j + 1] = y[j];
      j = j - 1;
    }
  }

  float a = x[i];
  float b = y[j];
  float c;
  while((i > 0) || (j > 0))
  {
    std::tie(c, e[i + j + 1]) = sum_err(a, b);
    a = c;

    if((i == 0) || ((j > 1) && (std::abs(y[j - 1]) < std::
       abs(x[i - 1]))))
    {
      b = y[j - 1];
      j = j - 1;
    }
    else
    {
      b = x[i - 1];
      i = i - 1;
    }
  }

  std::tie(c, e[1]) = sum_err(a, b);
  e[0] = c;

  return e;
}

/*
 * procedure renorm()
 *
 * Normalizes a floating-point number that is split into
 * multiple floating-point numbers contained in a vector.
 * This will compact the multiple floating-point numbers
 * as much as possible and return them in a new,
 * potentially smaller vector.
 */

std::vector<float> renorm(std::vector<float> &e)
{
  float c = e.back();
  std::vector<float> f(e.size());
  std::vector<float> s(e.size());

  for(int i = e.size() - 2; i >= 0; --i)
  {
    std::tie(c, f[i + 1]) = sum_err(c, e[i]);
  }

  f[0] = c;
  s[0] = f[0];
  int k = 0;
  float d;

  for(int j = 1; j < e.size(); ++j)
  {
    std::tie(c, d) = sum_err(s[k], f[j]);
    s[k] = c;

    if(d != 0.0)
    {
      k = k + 1;

      s[k] = d;
    }
  }

  s.erase(std::remove(s.begin(), s.end(), 0.0), s.end());
  if(s.empty())
  {
    s.push_back(0.0);
  }

  return s;
}

/*
 * procedure split()
 *
 * Splits a single floating-point number into 2 smaller,
 * normalized floating-point numbers that represent the
 * same numerical value as the original input when added
 * together.  The t argument represents the digit-count of
 * the input number and is generally the number of bits in
 * the mantissa (23 in the case of IEEE-754 single
 * precision).  The k argument represents the number of
 * 'nonzero digits' the first number returned should
 * contain.  For an near-even split (the usually-desired
 * case), this value should be t / 2 + 1.
 */

std::tuple<float, float> split(float x, int t, int k)
{
  float ak = std::pow(2, (t - k)) + 1;
  float y = ak * x;
  float z = y - x;
  float xp = y - z;
  float xpp = x - xp;

  return {xp, xpp};
}

/*
 * procedure multiply()
 *
 * Multiplies 2 floating-point numbers in the form of 2
 * vectors each containing multiple floating-point numbers
    .
 * The result is a new vector containing multiple
 * floating- * point numbers representing the product of
 * the inputs.
 */

std::vector<float> multiply(std::vector<float> &x, std::
    vector<float> &y)
{
#ifdef LIST_STEPS
  std::cout << "Starting (x * y) multiply with the
     following x and y values:" << std::endl;
  for(std::vector<float>::size_type i = 0; i < x.size();
     ++i)
  {
  std::cout << "\tx[" << i << "]:     " << x[i] << std::
     endl;
  }
  for(std::vector<float>::size_type i = 0; i < y.size();
     ++i)
  {
  std::cout << "\ty[" << i << "]:     " << y[i] << std::
     endl;
  }
#endif

  std::vector<float> xp(x.size());
  std::vector<float> xpp(x.size());
  std::vector<float> yp(y.size());
  std::vector<float> ypp(y.size());
  std::vector<float> yppp(y.size());
  std::vector<float> p(x.size() * y.size());

  for(int i = 0; i < x.size(); ++i)
  {
    std::tie(xp[i], xpp[i]) = split(x[i], 23, 23 / 2 + 1);

#ifdef LIST_STEPS
    std::cout << "Splitting x[" << i << "] into 2 parts:"
       << std::endl;
    std::cout << "\tx[" << i << "]:     " << x[i] << " ->"
       << std::endl;
    std::cout << "\txp[" << i << "]:    " << xp[i] << ","
       << std::endl;
    std::cout << "\txpp[" << i << "]:   " << xpp[i] << std
       ::endl;
#endif
  }
  for(int i = 0; i < y.size(); ++i)
  {
    float z;
    std::tie(yp[i], z) = split(y[i], 23, 23 / 3 + 1);
    std::tie(ypp[i], yppp[i]) = split(z, 23, 23 / 3 + 1);
```

```
#ifdef LIST_STEPS
    std::cout << "Splitting y[" << i << "] into 3 parts:"
       << std::endl;
    std::cout << "\ty[" << i << "]:     " << y[i] << " ->"
       << std::endl;
    std::cout << "\typ[" << i << "]:    " << yp[i] << ","
       << std::endl;
    std::cout << "\typp[" << i << "]:   " << ypp[i] << ","
       << std::endl;
    std::cout << "\typpp[" << i << "]: " << yppp[i] << std
       ::endl;
#endif
  }

  p[0] = 0;

  for(int i = 0; i < x.size(); ++i)
  {
#ifdef LIST_STEPS
    std::cout << "Iteration [" << i << "] of
       multiplication by parts:" << std::endl;
    std::cout << "\tMultiplication of individual parts:"
       << std::endl;
#endif
    std::vector<float> a1(y.size());
    std::vector<float> a2(y.size());
    std::vector<float> a3(y.size());
    std::vector<float> a4(y.size());
    std::vector<float> a5(y.size());
    std::vector<float> a6(y.size());

    for(int j = 0; j < y.size(); ++j)
    {
      a1[j] = xp[i] * yp[j];
      a2[j] = xp[i] * ypp[j];
      a3[j] = xp[i] * yppp[j];
      a4[j] = xpp[i] * yp[j];
      a5[j] = xpp[i] * ypp[j];
      a6[j] = xpp[i] * yppp[j];

#ifdef LIST_STEPS
      std::cout << "\ta1[" << j << "] = xp[" << i << "] *
         yp[" << j << "]:     " << a1[j] << std::endl;
      std::cout << "\ta2[" << j << "] = xp[" << i << "] *
         ypp[" << j << "]:    " << a2[j] << std::endl;
      std::cout << "\ta3[" << j << "] = xp[" << i << "] *
         yppp[" << j << "]:   " << a3[j] << std::endl;
      std::cout << "\ta4[" << j << "] = xpp[" << i << "] *
          yp[" << j << "]:    " << a4[j] << std::endl;
      std::cout << "\ta5[" << j << "] = xpp[" << i << "] *
          ypp[" << j << "]:   " << a5[j] << std::endl;
      std::cout << "\ta6[" << j << "] = xpp[" << i << "] *
```

```
            yppp[" << j << "]: " << a6[j] << std::endl;
#endif
    }

    std::vector<float> b;
    std::vector<float> c;
    std::vector<float> d;

    b = renorm(a1);
    c = renorm(a2);
    d = add(c, b);
    b = renorm(d);

#ifdef LIST_STEPS
    std::cout << "\tb = a1 + a2 = (renormalized)" << std::
       endl;
    for(std::vector<float>::size_type k = 0; k < b.size();
        ++k)
    {
    std::cout << "\tb[" << k << "]:  " << b[k] << std::
       endl;
    }
#endif

    c = renorm(a3);
    d = add(c, b);
    b = renorm(d);

#ifdef LIST_STEPS
    std::cout << "\tb = b + a3 = (renormalized)" << std::
       endl;
    for(std::vector<float>::size_type k = 0; k < b.size();
        ++k)
    {
    std::cout << "\tb[" << k << "]:  " << b[k] << std::
       endl;
    }
#endif

    c = renorm(a4);
    d = add(c, b);
    b = renorm(d);

#ifdef LIST_STEPS
    std::cout << "\tb = b + a4 = (renormalized)" << std::
       endl;
    for(std::vector<float>::size_type k = 0; k < b.size();
        ++k)
    {
    std::cout << "\tb[" << k << "]:  " << b[k] << std::
       endl;
    }
#endif
```

```
    c = renorm(a5);
    d = add(c, b);
    b = renorm(d);

#ifdef LIST_STEPS
    std::cout << "\tb = b + a5 = (renormalized)" << std::
       endl;
    for(std::vector<float>::size_type k = 0; k < b.size();
        ++k)
    {
    std::cout << "\tb[" << k << "]:  " << b[k] << std::
       endl;
    }
#endif

    c = renorm(a6);
    d = add(c, b);
    b = renorm(d);

#ifdef LIST_STEPS
    std::cout << "\tb = b + a6 = (renormalized)" << std::
       endl;
    for(std::vector<float>::size_type k = 0; k < b.size();
        ++k)
    {
    std::cout << "\tb[" << k << "]:  " << b[k] << std::
       endl;
    }
#endif

    d = add(b, p);
    p = renorm(d);

#ifdef LIST_STEPS
    std::cout << "\tp = b + p = (renormalized)" << std::
       endl;
    for(std::vector<float>::size_type k = 0; k < p.size();
        ++k)
    {
    std::cout << "\tp[" << k << "]:  " << p[k] << std::
       endl;
    }
#endif
  }

  return p;
}

/*
 * Main function for testing.
 */
```

```
int main()
{
  std::cout.precision(200);
  std::cout << std::fixed;

  std::vector<float> x;
  std::vector<float> y;
  std::vector<float> e;
  double ans;
  int x_bits;
  int y_bits;
  std::vector<int> e_bits;
  long ans_bits;

  // Print out single x value

  x.push_back(4.00000095367431640625);

  std::cout << "x value: " << std::endl;
  if(x[0] >= 0.0)
  {
    std::cout << " ";
  }
  std::cout << x[0] << std::endl;

  // Print hex of x value

  std::memcpy(&x_bits, &x[0], sizeof x[0]);
  std::cout << " 0x" << std::hex << x_bits << std::dec <<
     std::endl << std::endl;

  // Print out single y value

  y.push_back(2.000002384185791015625);

  std::cout << "y value: " << std::endl;
  if(y[0] >= 0.0)
  {
    std::cout << " ";
  }
  std::cout << y[0] << std::endl;

  // Print hex of y value

  std::memcpy(&y_bits, &y[0], sizeof y[0]);
  std::cout << " 0x" << std::hex << y_bits << std::dec <<
     std::endl << std::endl;

  // Multiply and print result

  e = multiply(x, y);

  std::cout << "Result of (x * y) in parts:" << std::endl;
  for(const auto &val : e)
  {
    if(val >= 0.0)
  {
    std::cout << " ";
  }
    std::cout << val << std::endl;

    int val_bits;
    std::memcpy(&val_bits, &val, sizeof val);
    e_bits.push_back(val_bits);
  }

  // Print hex of result

  for(const auto &val : e_bits)
  {
    std::cout << " 0x" << std::hex << val << std::dec <<
       std::endl;
  }
  std::cout << std::endl;

  // Show solution found using double precision

  ans = (double) x[0] * y[0];
  std::cout << "Result from double-precision calculation (
     x * y):" << std::endl;
  if(ans >= 0.0)
  {
    std::cout << " ";
  }
  std::cout << ans << std::endl;

  // Show hex of double precision solution

  std::memcpy(&ans_bits, &ans, sizeof ans);
  std::cout << " 0x" << std::hex << ans_bits << std::dec
     << std::endl;

  return EXIT_SUCCESS;
}
```

APPENDIX B: DOUBLE PRECISION C++ CODE EXAMPLE

```
/*
 * Algorithms for each function in this program were taken
 * from the paper: "Algorithms for Arbitrary Precision
 * Floating Point Arithmetic" by Douglas M. Priest.
 *
 * It is important to note that the original algorithms
 * are defined using a 1-base indexing system.  For this
 * implementation, they have been adjusted to use 0-base
 * indexing to match how C++ arrays work.
 *
 * Author: Alex Underwood
 *         alexander.underwood@okstate.edu
 */

// Remove this define to simply see the outputs and no
   intermediate steps.
//#define LIST_STEPS

#include <iostream>
#include <cstring>
#include <cmath>
#include <vector>
#include <algorithm>
#include <tuple>
#include <iomanip>
#include <quadmath.h>

/*
 * procedure sum_err()
 *
 * Calculates the sum of 2 floating point numbers and
 * returns the bits that exceed the size of the mantissa
 * as a second, normalized floating-point number.
 */

std::tuple<double, double> sum_err(double a, double b)
{
  if(std::abs(a) < std::abs(b))
  {
    double temp = a;
    a = b;
    b = temp;
  }

  double c = a + b;
  double e = c - a;
  double g = c - e;
  double h = g - a;
  double f = b - h;
  double d = f - e;

  if((d + e) != f)
  {
    c = a;
    d = b;
  }

  return {c, d};
}

/*
 * procedure add()
 *
 * Adds 2 floating-point numbers in the form of 2 vectors
 * each containing multiple floating-point numbers.  The
 * result is a new vector containing multiple floating-
 * point numbers representing the sum of the inputs.
 */

std::vector<double> add(std::vector<double> &x, std::
    vector<double> &y)
{
  int i = x.size() - 1;
  int j = y.size() - 1;
  std::vector<double> e(x.size() + y.size());

  if(std::abs(x[i]) < std::abs(y[j]))
  {
    while((i > 0) && (std::abs(x[i - 1]) <= std::abs(y[j])
        ))
    {
      e[i + j + 1] = x[i];
      i = i - 1;
    }
  }
  else if(std::abs(x[i]) > std::abs(y[j]))
  {
    while((j > 0) && (std::abs(y[j - 1]) <= std::abs(x[i])
        ))
    {
      e[i + j + 1] = y[j];
      j = j - 1;
    }
  }

  double a = x[i];
  double b = y[j];
  double c;
  while((i > 0) || (j > 0))
  {
    std::tie(c, e[i + j + 1]) = sum_err(a, b);
    a = c;

    if((i == 0) || ((j > 1) && (std::abs(y[j - 1]) < std::
       abs(x[i - 1]))))
    {
      b = y[j - 1];
      j = j - 1;
    }
    else
    {
      b = x[i - 1];
      i = i - 1;
    }
  }

  std::tie(c, e[1]) = sum_err(a, b);
  e[0] = c;

  return e;
}

/*
 * procedure renorm()
 *
 * Normalizes a floating-point number that is split into
 * multiple floating-point numbers contained in a vector.
 * This will compact the multiple floating-point numbers
 * as much as possible and return them in a new,
 * potentially smaller vector.
 */

std::vector<double> renorm(std::vector<double> &e)
{
  double c = e.back();
  std::vector<double> f(e.size());
  std::vector<double> s(e.size());

  for(int i = e.size() - 2; i >= 0; --i)
  {
    std::tie(c, f[i + 1]) = sum_err(c, e[i]);
  }

  f[0] = c;
  s[0] = f[0];
  int k = 0;
  double d;

  for(int j = 1; j < e.size(); ++j)
  {
    std::tie(c, d) = sum_err(s[k], f[j]);
    s[k] = c;

    if(d != 0.0)
    {
      k = k + 1;

      s[k] = d;
    }
  }

  s.erase(std::remove(s.begin(), s.end(), 0.0), s.end());
  if(s.empty())
  {
    s.push_back(0.0);
  }

  return s;
}

/*
 * procedure split()
 *
 * Splits a single floating-point number into 2 smaller,
 * normalized floating-point numbers that represent the
 * same numerical value as the original input when added
 * together.  The t argument represents the digit-count of
 * the input number and is generally the number of bits in
 * the mantissa (23 in the case of IEEE-754 single
 * precision).  The k argument represents the number of
 * 'nonzero digits' the first number returned should
 * contain.  For an near-even split (the usually-desired
 * case), this value should be t / 2 + 1.
 */

std::tuple<double, double> split(double x, int t, int k)
{
  double ak = std::pow(2, (t - k)) + 1;
  double y = ak * x;
  double z = y - x;
  double xp = y - z;
  double xpp = x - xp;

#ifdef LIST_STEPS
  std::cout << "\t\tsplit() function steps" << std::endl;
  std::cout << "\t\tak:  " << ak << std::endl;
  std::cout << "\t\ty:   " << y << std::endl;
  std::cout << "\t\tz:   " << z << std::endl;
  std::cout << "\t\txp:  " << xp << std::endl;
  std::cout << "\t\txpp: " << xpp << std::endl;
#endif

  return {xp, xpp};
}
```

```
/*
 * procedure multiply()
 *
 * Multiplies 2 floating-point numbers in the form of 2
 * vectors each containing multiple floating-point
 * numbers.  The result is a new vector containing
 * multiple floating-point numbers representing the
 * product of the inputs.
 */

std::vector<double> multiply(std::vector<double> &x, std::
    vector<double> &y)
{
#ifdef LIST_STEPS
  std::cout << "Starting (x * y) multiply with the
     following x and y values:" << std::endl;
  for(std::vector<double>::size_type i = 0; i < x.size();
     ++i)
  {
    std::cout << "\tx[" << i << "]:    " << x[i] << std::
       endl;
  }
  for(std::vector<double>::size_type i = 0; i < y.size();
     ++i)
  {
    std::cout << "\ty[" << i << "]:    " << y[i] << std::
       endl;
  }
#endif

  std::vector<double> xp(x.size());
  std::vector<double> xpp(x.size());
  std::vector<double> yp(y.size());
  std::vector<double> ypp(y.size());
  std::vector<double> yppp(y.size());
  std::vector<double> p(x.size() * y.size());

  for(int i = 0; i < x.size(); ++i)
  {
    std::tie(xp[i], xpp[i]) = split(x[i], 23, 23 / 2 + 1);

#ifdef LIST_STEPS
    std::cout << "Splitting x[" << i << "] into 2 parts:"
       << std::endl;
    std::cout << "\tx[" << i << "]:    " << x[i] << " ->"
       << std::endl;
    std::cout << "\txp[" << i << "]:   " << xp[i] << ","
       << std::endl;
    std::cout << "\txpp[" << i << "]:  " << xpp[i] << std
       ::endl;
#endif
  }
  for(int i = 0; i < y.size(); ++i)
  {
    double z;
    std::tie(yp[i], z) = split(y[i], 23, 23 / 3 + 1);
    std::tie(ypp[i], yppp[i]) = split(z, 23, 23 / 3 + 1);

#ifdef LIST_STEPS
    std::cout << "Splitting y[" << i << "] into 3 parts:"
       << std::endl;
    std::cout << "\ty[" << i << "]:     " << y[i] << " ->"
       << std::endl;
    std::cout << "\typ[" << i << "]:    " << yp[i] << ","
       << std::endl;
    std::cout << "\typp[" << i << "]:   " << ypp[i] << ","
       << std::endl;
    std::cout << "\typpp[" << i << "]: " << yppp[i] << std
       ::endl;
#endif
  }

  p[0] = 0;

  for(int i = 0; i < x.size(); ++i)
  {
#ifdef LIST_STEPS
    std::cout << "Iteration [" << i << "] of
       multiplication by parts:" << std::endl;
    std::cout << "\tMultiplication of individual parts:"
       << std::endl;
#endif
    std::vector<double> a1(y.size());
    std::vector<double> a2(y.size());
    std::vector<double> a3(y.size());
    std::vector<double> a4(y.size());
    std::vector<double> a5(y.size());
    std::vector<double> a6(y.size());

    for(int j = 0; j < y.size(); ++j)
    {
      a1[j] = xp[i] * yp[j];
      a2[j] = xp[i] * ypp[j];
      a3[j] = xp[i] * yppp[j];
      a4[j] = xpp[i] * yp[j];
      a5[j] = xpp[i] * ypp[j];
      a6[j] = xpp[i] * yppp[j];

#ifdef LIST_STEPS
      std::cout << "\ta1[" << j << "] = xp[" << i << "] *
         yp[" << j << "]:     " << a1[j] << std::endl;
      std::cout << "\ta2[" << j << "] = xp[" << i << "] *
         ypp[" << j << "]:    " << a2[j] << std::endl;
      std::cout << "\ta3[" << j << "] = xp[" << i << "] *
```

```
         yppp[" << j << "]:  " << a3[j] << std::endl;
      std::cout << "\ta4[" << j << "] = xpp[" << i << "] *
          yp[" << j << "]:   " << a4[j] << std::endl;
      std::cout << "\ta5[" << j << "] = xpp[" << i << "] *
          ypp[" << j << "]:  " << a5[j] << std::endl;
      std::cout << "\ta6[" << j << "] = xpp[" << i << "] *
          yppp[" << j << "]: " << a6[j] << std::endl;
#endif
    }

    std::vector<double> b;
    std::vector<double> c;
    std::vector<double> d;

    b = renorm(a1);
    c = renorm(a2);
    d = add(c, b);
    b = renorm(d);

#ifdef LIST_STEPS
    std::cout << "\tb = a1 + a2 = (renormalized)" << std::
       endl;
    for(std::vector<double>::size_type k = 0; k < b.size()
       ; ++k)
    {
      std::cout << "\tb[" << k << "]:  " << b[k] << std::
         endl;
    }
#endif

    c = renorm(a3);
    d = add(c, b);
    b = renorm(d);

#ifdef LIST_STEPS
    std::cout << "\tb = b + a3 = (renormalized)" << std::
       endl;
    for(std::vector<double>::size_type k = 0; k < b.size()
       ; ++k)
    {
      std::cout << "\tb[" << k << "]:  " << b[k] << std::
         endl;
    }
#endif

    c = renorm(a4);
    d = add(c, b);
    b = renorm(d);

#ifdef LIST_STEPS
    std::cout << "\tb = b + a4 = (renormalized)" << std::
       endl;
    for(std::vector<double>::size_type k = 0; k < b.size()
        ; ++k)
    {
      std::cout << "\tb[" << k << "]:  " << b[k] << std::
         endl;
    }
#endif

    c = renorm(a5);
    d = add(c, b);
    b = renorm(d);

#ifdef LIST_STEPS
    std::cout << "\tb = b + a5 = (renormalized)" << std::
       endl;
    for(std::vector<double>::size_type k = 0; k < b.size()
       ; ++k)
    {
      std::cout << "\tb[" << k << "]:  " << b[k] << std::
         endl;
    }
#endif

    c = renorm(a6);
    d = add(c, b);
    b = renorm(d);

#ifdef LIST_STEPS
    std::cout << "\tb = b + a6 = (renormalized)" << std::
       endl;
    for(std::vector<double>::size_type k = 0; k < b.size()
       ; ++k)
    {
      std::cout << "\tb[" << k << "]:  " << b[k] << std::
         endl;
    }
#endif

    d = add(b, p);
    p = renorm(d);

#ifdef LIST_STEPS
    std::cout << "\tp = b + p = (renormalized)" << std::
       endl;
    for(std::vector<double>::size_type k = 0; k < p.size()
       ; ++k)
    {
      std::cout << "\tp[" << k << "]:  " << p[k] << std::
         endl;
    }
#endif
  }

  return p;
```

```
}

/*
 * Main function for testing.
 */

int main()
{
  std::cout.precision(110);
  std::cout << std::fixed;

  std::vector<double> x;
  std::vector<double> y;
  std::vector<double> e;
  __float128 ans;
  long x_bits;
  long y_bits;
  std::vector<long> e_bits;
  __int128 ans_bits;

  // Print out single x value

  x.push_back(2.0000000000000004440892098500 6E0);

  std::cout << "x value: " << std::endl;
  if(x[0] >= 0.0)
  {
    std::cout << " ";
  }
  std::cout << x[0] << std::endl;

  // Print hex of x value

  std::memcpy(&x_bits, &x[0], sizeof x[0]);
  std::cout << " 0x" << std::hex << x_bits << std::dec <<
     std::endl << std::endl;

  // Print out single y value

  y.push_back(4.0000000000000017763568394002 5E0);

  std::cout << "y value: " << std::endl;
  if(y[0] >= 0.0)
  {
    std::cout << " ";
  }
  std::cout << y[0] << std::endl;

  // Print hex of y value

  std::memcpy(&y_bits, &y[0], sizeof y[0]);
  std::cout << " 0x" << std::hex << y_bits << std::dec <<
     std::endl << std::endl;
  // Multiply and print result

  e = multiply(x, y);

  std::cout << "Result of (x * y) in parts:" << std::endl;
  for(const auto &val : e)
  {
    if(val >= 0.0)
    {
      std::cout << " ";
    }
    std::cout << val << std::endl;

    long val_bits;
    std::memcpy(&val_bits, &val, sizeof val);
    e_bits.push_back(val_bits);
  }

  // Print hex of result

  for(const auto &val : e_bits)
  {
    std::cout << " 0x" << std::hex << val << std::dec <<
        std::endl;
  }
  std::cout << std::endl;

  // Show solution found using double precision

  ans = (__float128) x[0] * y[0];
  std::cout << "Result from double-precision calculation (
     x * y):" << std::endl;

  char buff[120];
  // Use Qe instead of Qg to get exponential output.
  quadmath_snprintf(buff, sizeof buff, "%*.110Qg", 46, ans
     );
  printf(" %s\n", buff);

  // Show hex of double precision solution

  long ans_parts[2];

  std::memcpy(&ans_parts, &ans, sizeof ans);
  std::cout << " 0x";
  std::cout << std::hex << std::setfill('0') << std::setw
     (16) << ans_parts[1] << std::dec;
  std::cout << std::hex << std::setfill('0') << std::setw
     (16) << ans_parts[0] << std::dec;
  std::cout << std::endl;

  return EXIT_SUCCESS;
}
```

VITA

Alex Underwood

Candidate for the Degree of

Master of Science

Thesis: IEEE FLOATING-POINT EXTENSION FOR MANAGING ERROR USING RESIDUAL REGISTERS

Major Field: Electrical Engineering

Biographical:

Education:

Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in December, 2019.

Completed the requirements for the Bachelor of Science in Computer Engineering at Oklahoma State University, Stillwater, Oklahoma in 2018.

Experience:

Graduate Research Assistant - VLSI Computer Architecture Research Group OSU
June 2018 - December 2019