IEEE Floating-Point Extension for Containing Error in the RISC-V Architecture

Alex Underwood, Tuan Nguyen and James E. Stine**
alexander.underwood,james.stine@okstate.edu
Oklahoma State University, VLSI Computer Architecture Research Group
Stillwater, Oklahoma, USA

ABSTRACT

This article discusses modifications to IEEE 754 floating-point units to help researchers and scientists monitor and control errors in scientific applications. 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. The use of the SoftFloat arithmetic validation suite is utilized and added to gem5. Simulation results are presented indicating 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 approximately 5% increase in area with less than 3% increase in energy over traditional IEEE 754 floating-point multipliers.

CCS CONCEPTS

• **Mathematics of computing → Arbitrary-precision arithmetic**; • **Hardware → Arithmetic and datapath circuits**.

KEYWORDS

IEEE 754 arithmetic, RISC-V floating-point, validated arithmetic

ACM Reference Format:
Alex Underwood, Tuan Nguyen and James E. Stine. 2019. IEEE Floating-Point Extension for Containing Error in the RISC-V Architecture. In *CARRV '19: Third Workshop on Computer Architecture Research with RISC-V, June 22, 2019, Phoenix, AZ*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Multiplication has long been an important part in most computer architectures and it has usually been utilized as a common case and as an architecture decision to include in any microarchitecture. However, the difficulty in creating hardware for multiplication because of its inherent shifting of the radix point has been a cogent reason for the need for floating-point hardware in scientific applications. To aid common usage for floating-point in computer architectures, the IEEE standardized floating-point in 1985 [1] and later re-ratified subsequently in 2008 [2] with the IEEE 754.

Although IEEE 754 floating-point implementations has made tremendous progress in making computations simpler and concise, it has an inherent problem within its structure. Since the dynamic range is much larger than normal integer and fractional implementations, it loses information when numbers are rounded to their final result. 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 [9], the precision must be preserved. Simply truncating denormal results to zero is unacceptable [13]. Consequently, having floating-point units that can handle normalized and denormalized numbers is essential, especially for scientific computing [14].

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 advantage in representable range makes it suitable for a wide range of applications [2]. However, recent research [5] 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 multipliers that can switch between precision numbers in implementing deep learning [6]. Moreover, it is important that the ability to do computations with larger precisions as well as monitor how much error computations exhibit.

To overcome the numerical limitations of existing computer systems, several software tools and hardware designs have been developed [24]. 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 and uses an extension to the method called native-pair computations [8] to the RISC-V architecture. Moreover, this paper also discusses architectural changes as well as simulation with the `gem5` architectural simulator [20].

*All authors contributed equally to this research.

CARRV '19, June 22, 2019, Phoenix, AZ

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

2 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 hardware as well as have an impact upon performance and cycle time. Over the years there has been many attempts to leverage hardware against simplicity yet still 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 [1].

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 [4], [15], and an overview of accurate vector arithmetic units is given in [3]. In addition, [15] presents carry-skip logic to determine if a carry-chain can be bypassed in the accumulator, based on a solution previously implemented in software [16]. 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 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 [1] and later amended in 2008 [2], defines the floating-point format that consists of three parts: sign (S), exponent (E), mantissa or significand (M). Figure 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 to allow users to support 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 shows a block diagram detailing the overall architecture. The design consists of several stages: unpack (hidden bit and other exception and bit testing), 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.

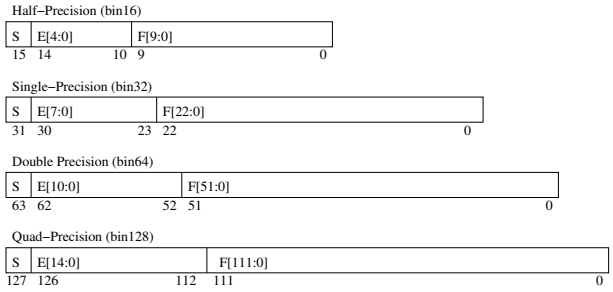

Figure 1: Data formats for the IEEE 754-2008 floating-point

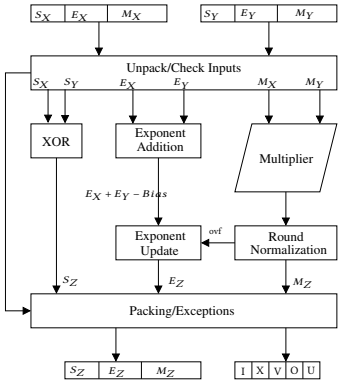

Figure 2: Block diagram of IEEE 754 multiplier architecture

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 places an important element in its use (e.g., conversion between integers and IEEE 754 arithmetic). Therefore, the need for architectures to be able to *analyze* error is important for high-performance computing and their applications.

A more pragmatic solution to this problem is utilizing something called native-pair arithmetic [8]. A pair of native floating-point is used to represent a base result and a residual term to increase accuracy is also computed. The original idea [8] 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 can possibly allow a group of several numbers to approximately double the amount of precision for a computation without having to add additional hardware [7]. As pointed out in [8], 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.

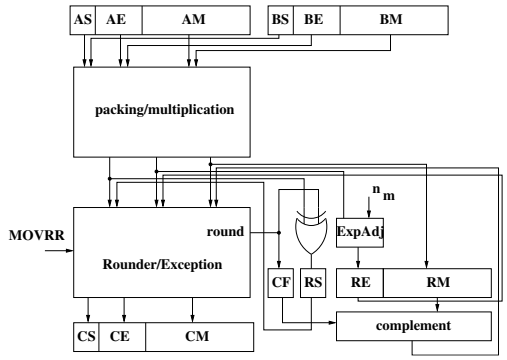

Figure 3: Previously Proposed Architecture for Residual Register in IEEE 754 Floating-Point Multiplier [8]

To accomplish this task, a residual register [8] 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, a new instruction is added to the Instruction Set Architecture (ISA) to handle moving it into the register file. The overall architecture looks like the architecture in Figure 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 [8]. Programs that do not use the residual register get the usual result defined by the IEEE 754 standard. Most 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 [8].

The architecture allows a good compromise between complexity and 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 [8]. Although the design in Figure 3 shows the change for IEEE 754 multiplication, the original idea in [8] can be applied to other IEEE 754 floating-point operations, as well.

The difficulty in rounding is due to the 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 [19]. 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 [9]. 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.

3 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., $P >= 2$) [23]. 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 [9]. Normally, compound adders use the same hardware except for critical components, such as the carry-chain logic [23].

RN is arguably the most complicated mode compared to RZ and RI modes. The method within [23] 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 [24].

Native-pair computations can be utilized to essentially build on top of current operations to create multi-precision computations [7, 12]. 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 [12]. 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].

As specified in [7], using multiple components and splitting their computations and accumulating them later is called native-pair floating-point computations in this paper, similar to [8]. It is argued in this paper that simpler architectural changes are needed that do not strangle other operations or *make the common case fast*. Although it is conceivable to perform this 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. Granted, this operation, would consumed 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 [8], 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

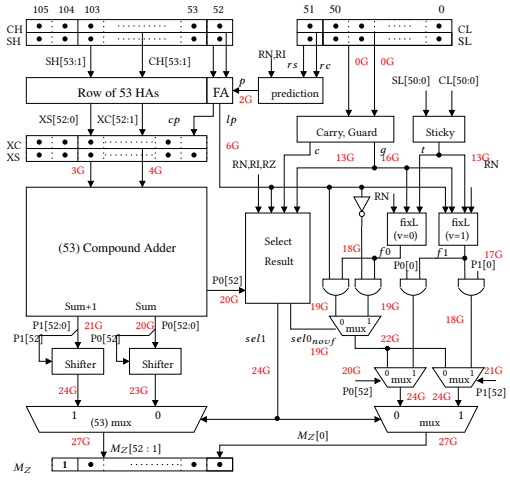

Figure 4: Rounding architecture for all IEEE 754 rounding modes (Adapted from [18])

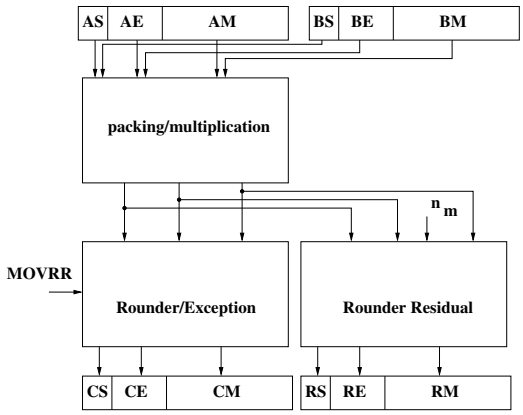

Figure 5: Proposed Architecture for Residual Register in IEEE 754 Floating-Point Multiplier

2) for the mantissa domain $[1, 2)$ by a right shift if it is equal to or larger than 2. The current implementation in [8] does not use current architectures that well known for IEEE 754 floating-point architectures [10, 22, 23]. This research has shown good architectures to optimize one of the main delay issues within IEEE 754 multiplication, the rounder. Recent research [17, 18] has given further optimizations into this critical part by analyzing each design. This optimized rounder can be seen in Figure 4.

Figure 4 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 from or `SH`, `CH` and the 52 least-significant bits 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 4 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 possible outputs `P0`, `P1` [18]. Both `P0` and `P1` are normalized before the final selection logic. On the right side of Figure 4, the carry `c`, guard `g`, and sticky `t` bits are computed based on `SL` and `CL` bits [18]. 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 4 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 [9]. 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 4, the normalization signal, `sel1` set by the **Select Result** unit, consumes 24G delays. This signifies that the logic in Figure 3 requires a significant amount of delay before the residual register can be computed. As seen in Figure 4, 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 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 5) 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 additional infrastructure within the microarchitecture to handle additional outputs from IEEE 754 FPUs.

To provide and example of how this implementation works, an example is given for native-pair multiplication based on the work in [21]. C++ programs are 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. Therefore, 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 [8].

For example, given two double 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 = 4.00000000000000017763568394002505 ...
    4646778106689453125
  = 0x4010_0000_0000_0002
y = 2.00000000000000004440892098500626 ...
    6161694526672363281 25
  = 0x4000_0000_0000_0001
```

```
z = x * y
z[1] = 8.000000000000005329070518200 ...
       751394033432006835937 5
     = 0x4020_0000_0000_0003
z[2] = 0.000000000000000000000000000 ...
       000788860905221011805411728 56 ...
       528278622967320643510902300 47 ...
       702789306640625
     = 0x39b0_0000_0000_0000
```

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

```
z = (quad) x * y
z = 8.000000000000005329070518200752 ...
    182894337227847742911728565282 78 ...
    622967320643510902300477027893 06 ...
    640625
  = 0x4002_0000_0000_0000_3000_0000_0000_0200
```

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 [24], 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.

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 2. 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 1.

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 [11], 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 [14]. An additional instruction 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 in Table 1, 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 [1, 2], 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.

The modifications provided in this work do not incur any extra modifications 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 [11] 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).

4 CONCLUSION

This work demonstrates work with the RISC-V architecture to aid in controlling accuracy and precision within IEEE 754 floating-point units. The architecture is designed to achieve small additions to floating-point instructions with a new instruction within the ISA. An additional floating-point validation mechanism is inserted

SPEC CPU Benchmark	444.namd	470.lbm	508.namd_r	519.lbm_r	619.lbm_s	644.nab_s
Runtime Information						
Simulated Seconds	17.55132	10.259652	10.034215	1482.954635	75.938858	2.635289
Real Seconds Elapsed	101470.81	28745.83	67628.68	4778013.15	220672.45	17830.05
# of Simulated Cycles	35102640085	20519304925	20068429170	2965909270356	151877716470	5270577837
Function Frequency						
Total Function Calls	47413825438	6610717022	34198662665	1595256157701	52256329490	8446078443
FloatADD	5917003819	2273240080	3743658184	541500665712	15594087856	990094464
FloatMULT	4414971460	1273446080	3294084157	326750716512	9024897856	1192676971
% of Runtime						
FloatADD	12.48%	34.39%	10.95%	33.94%	29.84%	11.72%
FloatMULT	9.31%	19.26%	9.63%	20.48%	17.27%	14.12%

Table 1: Results of RISC-V gem5 simulations

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 2: gem5 Simulation Specifications

within `gem5` utilizing the `SoftFloat` software library. This new addition to the `gem5` simulator allows easy simulation of correct IEEE 754 floating-point arithmetic. It is anticipated that this new addition along with modifications to the `gem5` with `SoftFloat` will be available for download to users at the workshop. The new addition of this software library can allow changes to IEEE 754 floating-point arithmetic and allow designers to test new architectures that might affect precision, such as with machine-learning.

REFERENCES

[1] 1985. IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985* (1985), 1–14. https://doi.org/10.1109/IEEESTD.1985.82928

[2] 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (Aug 2008), 1–70. https://doi.org/10.1109/IEEESTD.2008.4610935

[3] G. Bohlender. 1990. What Do We Need Beyond IEEE Arithmetic? In *Computer Arithmetic and Self-Validating Numerical Methods*, Ch. Ullrich (Ed.). Academic Press, 1–32.

[4] P. R. Capello and W. L. Miranker. 1988. Systolic super summation. *IEEE Trans. Comput.* 37, 6 (June 1988), 657–677. https://doi.org/10.1109/12.2205

[5] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Low precision arithmetic for deep learning. *CoRR* abs/1412.7024 (2014). http://arxiv.org/abs/1412.7024

[6] J. Dean, D. Patterson, and C. Young. 2018. A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution. *IEEE Micro* 38, 2 (Mar 2018), 21–29. https://doi.org/10.1109/MM.2018.112130030

[7] T. Dekker. 1971. A Floating-Point Technique for Extending the Available Precision. *Numer. Math.* 18 (1971), 224–242.

[8] W. R. Dieter, A. Kaveti, and H. G. Dietz. 2007. Low-Cost Microarchitectural Support for Improved Floating-Point Accuracy. *IEEE Computer Architecture Letters* 6, 1 (Jan 2007), 13–16. https://doi.org/10.1109/L-CA.2007.1

[9] Milo D. Ercegovac and Tomas Lang. 2003. *Digital Arithmetic* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[10] Guy Even and P-M Seidel. 2000. A comparison of three rounding algorithms for IEEE floating-point multiplication. *IEEE Trans. Comput.* 49, 7 (2000), 638–650.

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

[12] Y. Hida, X. S. Li, and D. H. Bailey. 2001. Algorithms for quad-double precision floating point arithmetic. In *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001.* 155–162. https://doi.org/10.1109/ARITH.2001.930115

[13] Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (second ed.). Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898718027 arXiv:http://epubs.siam.org/doi/pdf/10.1137/1.9780898718027

[14] W. Kahan. 1996. *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic.* Technical Report. University of California, Berkeley. Available at http://www.cs.berkeley.edu/~wkahan.

[15] A. Knofel. 1991. Fast hardware units for the computation of accurate dot products. In *[1991] Proceedings 10th IEEE Symposium on Computer Arithmetic.* 70–74. https://doi.org/10.1109/ARITH.1991.145536

[16] M. Muller, C. Rub, and W. Rulling. 1991. Exact accumulation of floating-point numbers. In *[1991] Proceedings 10th IEEE Symposium on Computer Arithmetic.* 64–69. https://doi.org/10.1109/ARITH.1991.145535

[17] T. D. Nguyen, S. Bui, and J. E. Stine. 2018. 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).* 1–8. https://doi.org/10.1109/ASAP.2018.8445092

[18] T. D. Nguyen, S. R. Thompson, and J. E. Stine. 2018. Architectural Improvements in IEEE-compliant Floating-Point Multiplication. *submitted to IEEE Transactions on Computers* (2018).

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

[20] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood. 2015. gem5-gpu: A Heterogeneous CPU-GPU Simulator. *IEEE Computer Architecture Letters* 14, 1 (Jan 2015), 34–36. https://doi.org/10.1109/LCA.2014.2299539

[21] D. M. Priest. 1991. Algorithms for arbitrary precision floating point arithmetic. In *[1991] Proceedings 10th IEEE Symposium on Computer Arithmetic.* 132–143. https://doi.org/10.1109/ARITH.1991.145549

[22] Nhon T Quach, Naofumi Takagi, and Michael J Flynn. 2004. Systematic IEEE rounding method for high-speed floating-point multipliers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12, 5 (2004), 511–521.

[23] M. R. Santoro, G. Bewick, and M. A. Horowitz. 1989. Rounding algorithms for IEEE multipliers. In *Proceedings of 9th Symposium on Computer Arithmetic.* 176–183. https://doi.org/10.1109/ARITH.1989.72824

[24] M. J. Schulte and E. E. Swartzlander, Jr. 1996. *Software and Hardware Techniques for Accurate, Self-Validating Arithmetic.* Kluwer Academic Publishers. 381–404 pages.