Implementation of a RISC-V compatible Multiply-Add Fused Unit

Tzu-Han Hsu
dept. Computer Science
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
hankshyu@gmail.com

***Abstract*—The floating-point Multiply-Add Fused (MAF, also known as Multiply-ACcumulate, MAC) unit is popular in modern microprocessor design due to its efficiency and performance advantages. The design aims to speed up scientific computations, multimedia applications, and in particular, convolutional neural networks for machine learning tasks. This study implements a MAF unit with RISC-V "F" extension compatibility [1], incorporating standard IEEE 754-2008 [2] exception handling, NaN propagation, and denormalized number support. Five distinct rounding modes and accrued exception flags are also supported in the proposed design. We test our implementation with carefully crafted corner cases and random generated floating-point numbers to verify its correctness. Source code of the project is available on https://github.com/hankshyu/RISC-V_MAC.**

***Index Terms*—Floating-Point Unit, Multiply-Add fused, Multiply Accumulate, RISC-V**

I. Introduction

Floating-point operations play a crucial role in modern day computing, especially when the machine learning domain flourishes. The growing computational power makes training sophisticated models possible. To apply the machine learning models in real life applications typically requires floating-points computations, which is demanding since large amount of real time data must be processed. Moreover, deep learning algorithms with exhaustive need of floating-point computational capabilities, such as neural networks, grew its popularity recently. These applications further challenge the floating-point processing power of the microprocessors. Among all floating-point operations, add-and-multiply are the most demanding one, the combination appears in the convolution layers of convolutional neural networks, digital filtering, and many other computing models' architecture.

Floating-point units are available on most microprocessors nowadays. Most designs center around a fused multiply-add dataflow due to its simplicity and performance advantage over separate multiplier and adder pipelines. It combines two basic operations with only one rounding error and shares hardware components to save chip area. Such design is also consistent with the basic RISC philosophy of heavily optimize key units in order to rapidly carry out the most frequently expected functions. Furthermore, the existence of fused multiply-add unit leads to more efficient superscalar CPU design since three floating-point instructions: add, multiply, and fused multiply-add could be scheduled to the same functional unit.

To take full advantage of the MAF dataflow, [3] transforms a set of equations into a series of multiply-adds by a numerical analysis technique called Horner's rule. [4] presents a general method to convert any transform algorithm into MAF optimized algorithms. [5] presents a framework for automatically generating MAF code for every linear DSP transform. The above-mentioned examples shows that the MAF architecture is recognized in modern computing and could receive optimization at the software level.

II. Related Work

IBM RS/6000 workstation [6] in 1990 is the first processor to contain a fused multiply-add dataflow. The compact design with pipeline latency of two cycles is made possible by a fast-shifter and a novel leading-zero/one anticipator [7]. It brought the concept of fusing multiply and add operation to reduce port number, and for better compiler optimization. The design is so classic that many of the hardware implementation algorithms used in RS/6000 are still popular today.

Later, IBM released a high-performance microprocessor optimized for commercial workloads called z990 eServer [8], also the first IBM mainframe equipped with a fused multiply-add floating-point unit [9]. It supports both the zSeries hexadecimal floating-point architecture [10] and the IEEE 754 binary floating-point architecture [11]. Furthermore, the instructions are executed in five pipeline stages. z990 eServer is not only famous for its variety of predecessors [12]–[17], but also for its aim to optimize the binary floating-point unit and have a fast multiply-add execution workflow.

As more microprocessors start to incorporate the ingenious MAF concept into their design, innovations for improvements also mushroomed. [18] supports multiple IEEE precisions MAF with single instruction multiple data (SIMD). The datapath is designed to execute one double-precision or two parallel single precision operations with around 20% more area

This is a undergraduate thesis project final report. I'm grateful to my advisor, Professor Chun-Jen Tsai for guiding and supporting me.

and with 10% more delay. Standard operations of floating-point add and multiply are performed by the MAF unit by setting multiplier 1 and addend 0. [19] proposes an architecture permitting skipping pipeline stages to speed up the operation. The design could potentially save 2 to 3 cycles when executing a floating-point add instruction.

Floating-point computation often require high implementation cost in hardware, hence microcontroller-level processors often carry out calculations with software emulation. By replacing Floating-point operations with library function call at compile time, software emulation associates with long computation time, compromised efficiency and large memory usage. [20] designed an area-optimized IEEE 754 compliant RISC-V Floating-point unit with MAF dataflow specifically for area sensitive microcontrollers, achieving 15 times speedups when compared with software emulation.

III. Architecture

A fused multiply-add can be described as the following equation:

$$Answer = A + B * C \tag{1}$$

A is called the addend, B is the multiplicand and C is the multiplier. The input operands are in IEEE 754 format and are further split into the sign bit, the exponent and the mantissa portion without the leading number of 1. The SpecialCaseDetector checks if the input value represents a zero, a NaN, an infinity, or a denormalized number. Special values may require special treatments in the datapath, so detecting them in the early stage is necessary.

The mantissa of the multiplier and the multiplicand would be sent to the Multiplier for product calculation while the mantissa of the addend would be the input of the PreNormalizer. The PreNormalizer and the Exponent Processor reads the exponents and calculates the amount of shifts necessary for the PreNormalizer to align the product and the addend. After the alignment, the shifted addend would be split into two parts, the lower part would join the CarrySaveAdder with the product whilst the upper part would be sent to the MSBIncrementer. Whether to increment the upper part or not would be decided by the output result of the EACAdder.

After concatenate the result from the EACAdder and the MSBIncrementer, an unnormalized answer is formed. The LeadingOneDetector scans the answer to determine how many bits of shifts does the Normalizer need to normalize the answer. Rounder would accept the normalized answer and deal with exception handling procedures if necessary. Flags will be raised at this stage too. Before the final answer is presented, the rounder would then adjust the answer with the rounding mode specified by the user.

The design of the multiplier involves creating a partial product array made up of multiples of multiplicand (R4Booth), and sum them up to form the product (WallaceTree). One of the key factors in designing a fast multiplier is to determine the radix. Choosing a smaller radix creates loads of partial products that is easy to calculate and pick from, but harder to sum due to the large quantity. On the other hand, a larger radix generates fewer partial products, but it's much more difficult to create the partial product array.

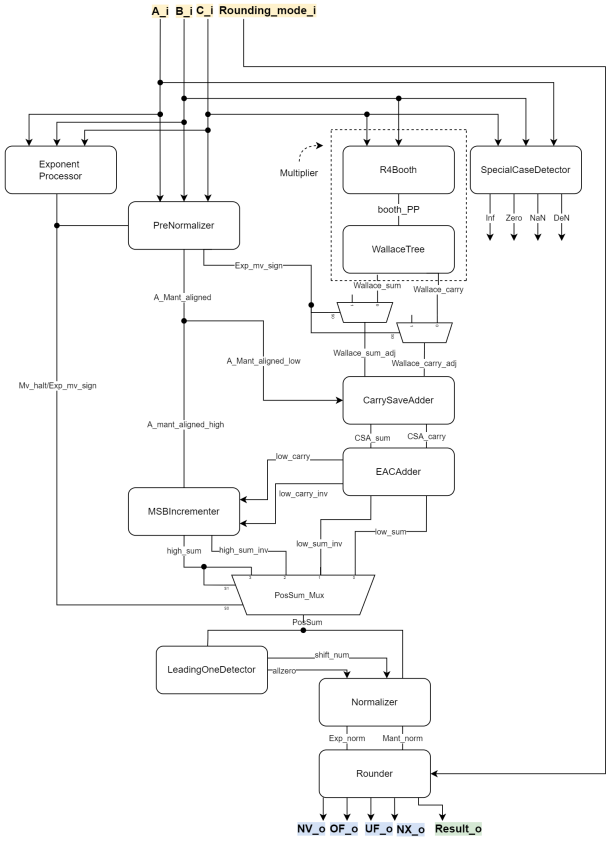

Fig. 1. Overall MAF Unit Architecture

Radix-10 multiplication is what we are most familiar with, often carry out by hand. Decimal format is optimal for financial applications and may become more popular in the future by the publication of the revision IEEE 754 floating-point standard [2]. Binary is yet the choice for most designers for the sake of its mathematical properties and performance advantage. Assume that both the multiplier and the multiplicand have N bit. Radix-2, the most naive binary, would require a large counter tree to sum up N partial products. Radix-4 multiplication will reduce the number of partial product to $ceil((N+1)/2)$, halves the number of radix-2. The downside is the partial product ranges from 0x 1x 2x 3x the multiplicand, 3x multiple may require extra delay and area to form since it is non-trivial.

Booth showed a technique to record digits in both positive and negative. Such transformation eliminates two consecutive ones thus eliminates the 3x multiple. A Booth radix-4 scanning simplifies the multiples to signed 0x 1x and 2x. Under the proposed implementation, the scanning process involves examining 3 bits of the multiplier, compare it to the Modified Booth's Recording Table and determine the

TABLE I
MODIFIED BOOTH'S RECORDING TABLE

bit			Partial
i+1	i	i-1	*Product*
0	0	0	+0 * Multiplicand
0	0	1	+1 * Multiplicand
0	1	0	+1 * Multiplicand
0	1	1	+2 * Multiplicand
1	0	0	-2 * Multiplicand
1	0	1	-1 * Multiplicand
1	1	0	-1 * Multiplicand
1	1	1	-0 * Multiplicand

multiplicand selected. The logic could be simplified to the equation below:

$$mul1x_o = bit_i \oplus bit_{i-1} \tag{2}$$

$$mul2x_o = bit_{i+1}\overline{bit_i bit_{i-1}} \vee \overline{bit_{i+1}} bit_i bit_{i-1} \tag{3}$$

$$mulsign_o = bit_{i+1} \tag{4}$$

The fact that multiplicand may be negative is disturbing because we must sign-extend the multiplicand during calculations. The intuitive way is by sign-extending every single bit on the left of the partial product. [21] mentioned an elegant way of acquiring the correct result while the sign of the partial product only effects two bits of the partial product, greatly improves the potential wiring of the design and is adopted in the proposed multiplier.

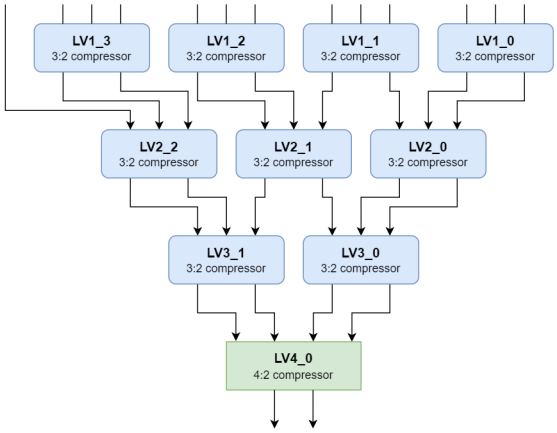

Fig. 2. Wallace tree structure implemented in our design

The next step of multiplication is to sum the partial products, a hardware structure named Wallace tree is implemented to reduce the critical path. Wallace trees are usually composed of carry save adders, also called counters. Traditionally, they intake 3 partial sums to output a sum and carry. 4:2 counters were introduced in [22] and were further optimized and designed [23] [24]. By mixing the use of 3:2 counter and 4:2 counter results in an optimal design which minimizes delay.

A. PreNormalizer

Before the Addend could be add with the partial products from the multiplier, proper alignment must take place. Our design overlaps the data alignment with the early phases of multiplication, such design requires shifting the addend in either direction while each of these partial products are two times as wide as the input. In short, by executing multiplication and alignment in parallel, we must include a large shifter about 3 times the size of the mantissa.

In a normal floating-point adder, the smaller exponent is aligned; nevertheless, it is very costly to implement a large shifter capable of shifting bidirectionally. A clean and more efficient implementation is mentioned in [25], alignment of the addend is implemented by placing the addend to the leftmost of the product and shift to the right according to the exponent value. In other words, the product is treated as having a fixed radix point and the addend is aligned to the radix point. Under such implementation, the shifting ranges approximately three times the width of the data plus some guard bits but only shift-right capabilities is needed for the shifter.

If the mantissa of the addend is length of the fraction plus two guard bits greater than the product, the answer is solely decided by the product's mantissa thus further shifting to the left is unnecessary. When the addend's most significant bit is less than the product's least significant bit, the mantissa of the addend dominates the result so it's needless to further shift to the right. The value of shifting is mentioned in [25] and is processed in the Exponent Processor in our design. Variable d is the difference between the exponent of the addend and the product, mv represents the shift amount of the addend for alignment:

$$d = expA - (expB + ExpC - 127) \tag{5}$$

$$mv = 27 - d \tag{6}$$

B. End Around Carry (EAC) Adders

In an ordinary Multiply-and-add dataflow, the product should form before the add operation takes place. However, the addend is summed with the partial products coming out of the multiplier directly in our design. Such dataflow may cause the propagation of any carry outs that ought to be ignored, contaminating the final result. In other words, we should figure out whether there was a carry out of the sign extension prior to the last carry save adder, if the carry out is detected, no adjustment has to be made. On the other hand, if no carry out is detected, we must invert the result driving to the EACAdders.

Floating-points are represented as the sign and magnitude format in IEEE 754. In consequence, the adder is only responsible of calculating the absolute value of the sum. However, it is very difficult to determine in advance which operand is bigger in the MAF dataflow. Even if we know which operand is greater, we still need two complementors for the sum and carry coming from the CarrySaveAdder, which is inefficient. We demand an adder that always output the magnitude of

the result by conditionally complement one operand, which is called an "End Around carry" adder.

The mathematical model is presented in [26]. The logic is essentially driving the carry out of (P-B) into another adder's carry in. Such function could be implemented by two carry chains, using two adders calculating (P - B) and (B - P) respectively with a multiplexer selecting the answer of the two. [26] also mentioned another implementation similar to a carry look-ahead adder, which is smaller in size but harder to implement.

C. Leading One Detector

Design of the Leading One detector is pivotal to the normalization process. Normalization strips away all leading sign bits so that the two bits adjacent to the radix point are of opposite polarity. To determine how many bits to shift would be the responsibility of the Leading one detector. Back to the first processor contains a fused multiply-add dataflow [6], the RC/6000 processor also equipped a leading-zero anticipator (LZA) to process the leading zeros and ones in parallel with floating-point addition. The algorithm is mentioned in [7], [27]. [28] further compares algorithms of detecting leading zeros/ones.

Although running one detection and addition in parallel would accelerate the calculation, the hardware area grows significantly if the input bits grew wider. Another disadvantage of calculating the leading ones beforehand is that the polarity of the addition is not yet determined. The hardware must incorporate the sign of the sum to calculate the correct number of leading ones. An easy solution is to only implement parts of the LZA component. Despite the fact that it would only operate when the sum is calculated, leading to a slower design. The lightweight leading one detector could assume the input is always positive, since the output of the EACAdder is always positive. By taking the advantage of the know polarity, our design uses a much smaller area and a simpler algorithm to achieve the correct result with great scalability.

D. Rounder

Since a floating-point number has a fixed sized mantissa, bits that are less significant would be naturally truncated during the calculation. For user to freely select their desired rounding mode, we must add extra bits named guard bit, round bit and sticky bit during the calculation. [29] explains why a guard bit is necessary to ensure the rounding works correctly.

RISC-V "F" Standard Extension [1] supports 5 rounding modes: RNE, RTZ, RDN, RUP and RMM. IEEE 754-2008 [2] clearly defines the behavior of rounding toward a directed orientation, which is how RTZ, RDN and RUP operates. [25] provided an easy way of implementation, simplifies 3 rounding modes into two: RI and RZ. RNE and RMM are a bit trickier. If the distance between two nearest floating-point numbers are equal, RNE delivers the one with even least significant digit, where RMM delivers the one with larger magnitude. They are named as roundTiesToEven and roundTiesToAway in IEEE 754-2008.

The floating-point control and status register in RISC-V also holds the accrued exception flags, NV, DZ, OF, UF and NX. In the MAC dataflow, DZ would never be raised so only 4 exception flags are handled in the proposed design. Overflow and underflow flags are pretty intuitive, NV flag will be raised if any invalid operation takes place. IEEE 754-2008 7.2 [2] lists invalid operations that shall be detected. NX flag stands for inexact, the exception would fire if the calculated result of our design does not equal to the absolute answer. By checking the contamination of the sticky bits, we could judge whether the flag shall be raised.

IEEE 754-2008 also defines the default exception handling methods that we must obey. Exception handling is also done by the Rounder because rounding mode could potentially affect the way underflow or overflow represents. For example, RTZ carries positive overflow to the format's largest finite number while RUP carries to the positive infinity. After all the adjustments, the output from the rounder drives the output of the proposed MAC module.

IV. Results

A testbench program is carefully designed to test the module. Most of the test-cases are hand-carved in order to come up with the error prone border test-cases. we have tested our module with over 100 border test-cases with different rounding modes. The test is written with a SystemVerilog testbench file. The test is split into several parts:

1) Invalid operations: Test if the module's exception handling works.
2) Infinities: Test if any infinity operand works as defined in IEEE 754-2008.
3) Operation with zeros: Test if the operations with zero work properly.
4) Overflow/Underflow: Test if the flags raise correctly and the output value adjusted according to the rounding mode.
5) Rounding: Test whether the five supported rounding modes work as expected.
6) Denormalized Numbers: Test if the denormalized numbers are fully supported; works if input/output is denormalized.
7) Random generated numbers: Generate random floating-point numbers to test if MAC operates as the way we want.

Our module passes all of the above-mentioned tests using a waveform simulator. We plan to further test our module with exhausting torture tests and on development boards in the future.

V. Conclusion

In this paper, we implemented a RISC-V "F" Extension compatible fused Multiply-Add Unit. The multiplier incorporates the radix-4 booth algorithm to generate partial products and with the crafted Wallace tree to reduce the critical path. The End Around Carry Adders is a more efficient way to sum the lower part of the addend with the product, which is

used in our design. Although there are faster algorithms out there, our implementation of Leading One detector is much more scalable and balanced in chip area. The Rounder is responsible of handling exceptions, raise flags and produce the rounded result as the answer. We have verified our design with a waveform simulator and testbenches for exception handling capabilities and corner cases.

The design of the fused multiply-add dataflow has changed since its first presentation in 1990, yet it still influences the designs of the floating-point units in microprocessors. The research on the development of MAF units has not yet stop. Researches of more compact counter trees and optimized addition path are still in progress. Hopefully, a more powerful floating-point unit with a renewed MAF dataflow would meet the future demand for floating-point computing.

REFERENCES

[1] "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213", Editors Andrew Waterman and Krste Asanovic, RISC-V Foundation, December 2019.

[2] "IEEE Standard for Floating-Point Arithmetic," in IEEE Std 754-2008 , vol., no., pp.1-70, 29 Aug. 2008, doi: 10.1109/IEEESTD.2008.4610935.

[3] Knuth, D. "The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, 3rd ed." Addison-Wesley, Reading, MA, 1998, 467–469.

[4] Y. Voronenko and M. Puschel, "Mechanical Derivation of Fused Multiply–Add Algorithms for Linear Transforms," in IEEE Transactions on Signal Processing, vol. 55, no. 9, pp. 4458-4473, Sept. 2007, doi: 10.1109/TSP.2007.896116.

[5] Y. Voronenko and M. Puschel, "Automatic generation of implementations for DSP transforms on fused multiply-add architectures," 2004 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2004, pp. V-101, doi: 10.1109/ICASSP.2004.1327057.

[6] R. K. Montoye, E. Hokenek and S. L. Runyon, "Design of the IBM RISC System/6000 floating-point execution unit," in IBM Journal of Research and Development, vol. 34, no. 1, pp. 59-70, Jan. 1990, doi: 10.1147/rd.341.0059.

[7] E. Hokenek and R. K. Montoye, "Leading-zero anticipator (LZA) in the IBM RISC System/6000 floating-point execution unit," in IBM Journal of Research and Development, vol. 34, no. 1, pp. 71-77, Jan. 1990, doi: 10.1147/rd.341.0071.

[8] T. J. Slegel, E. Pfeffer and J. A. Magee, "The IBM eServer z990 microprocessor," in IBM Journal of Research and Development, vol. 48, no. 3.4, pp. 295-309, May 2004, doi: 10.1147/rd.483.0295.

[9] G. Gerwig et al., "The IBM eServer z990 floating-point unit," in IBM Journal of Research and Development, vol. 48, no. 3.4, pp. 311-322, May 2004, doi: 10.1147/rd.483.0311.

[10] IBM Corporation, Enterprise Systems Architecture/390 Principles of Operation (SA22-7201); see http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi/.

[11] "IEEE Standard for Binary Floating-Point Arithmetic," in ANSI/IEEE Std 754-1985 , vol., no., pp.1-20, 12 Oct. 1985, urldoi: 10.1109/IEEESTD.1985.82928.

[12] G. Gerwig and M. Kroener, "Floating-point unit in standard cell design with 116 bit wide dataflow," Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No.99CB36336), 1999, pp. 266-273, doi: 10.1109/ARITH.1999.762853.

[13] E. M. Schwarz, L. Sigal and T. J. McPherson, "CMOS floating-point unit for the S/390 Parallel Enterprise Server G4," in IBM Journal of Research and Development, vol. 41, no. 4.5, pp. 475-488, July 1997, doi: 10.1147/rd.414.0475.

[14] E. M. Schwarz, R. M. Averill and L. J. Sigal, "A radix-8 CMOS S/390 multiplier," Proceedings 13th IEEE Sympsoium on Computer Arithmetic, 1997, pp. 2-9, doi: 10.1109/ARITH.1997.614873.

[15] E. M. Schwarz and C. A. Krygowski, "The S/390 G5 floating-point unit," in IBM Journal of Research and Development, vol. 43, no. 5.6, pp. 707-721, Sept. 1999, doi: 10.1147/rd.435.0707.

[16] E. M. Schwarz, R. M. Smith and C. A. Krygowski, "The S/390 G5 floating point unit supporting hex and binary architectures," Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No.99CB36336), 1999, pp. 258-265, doi: 10.1109/ARITH.1999.762852.

[17] E. M. Schwarz et al., "The microarchitecture of the IBM eServer z900 processor," in IBM Journal of Research and Development, vol. 46, no. 4.5, pp. 381-395, July 2002, doi: 10.1147/rd.464.0381.

[18] Chichyang Chen, Liang-An Chen and Jih-Ren Cheng, "Architectural design of a fast floating-point multiplication-add fused unit using signed-digit addition," Proceedings Euromicro Symposium on Digital Systems Design, 2001, pp. 346-353, doi: 10.1109/DSD.2001.952324.

[19] T. Lang and J. D. Bruguera, "Floating-point fused multiply-add with reduced latency," Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2002, pp. 145-150, doi: 10.1109/ICCD.2002.1106762.

[20] Bertaccini, Luca & Perotti, Matteo & Mach, Stefan & Schiavone, Pasquale & Zaruba, Florian & Benini, Luca. (2021). Tiny-FPU: Low-Cost Floating-Point Support for Small RISC-V MCU Cores. 1-5. 10.1109/ISCAS51556.2021.9401149.

[21] Bewick, Gary & Flynn, Michael. (1970). Fast Multiplication: Algorithms And Implementation.

[22] Weinberger, A. "4-2 carry-save adder module." IBM technical disclosure bulletin 23.8 (1981): 3811-3814.

[23] D. Radhakrishnan and A. P. Preethy, "Low power CMOS pass logic 4-2 compressor for high-speed multiplication," Proceedings of the 43rd IEEE Midwest Symposium on Circuits and Systems (Cat.No.CH37144), 2000, pp. 1296-1298 vol.3, doi: 10.1109/MWSCAS.2000.951453.

[24] K. Prasad and K. K. Parhi, "Low-power 4-2 and 5-2 compressors," Conference Record of Thirty-Fifth Asilomar Conference on Signals, Systems and Computers (Cat.No.01CH37256), 2001, pp. 129-133 vol.1, doi: 10.1109/ACSSC.2001.986892.

[25] Zhaolin Li, Xinyue Zhang, Gongqiong Liz and Runde Zhou, "Design of a fully pipelined single-precision floating-point unit," 2007 7th International Conference on ASIC, 2007, pp. 60-63, doi: 10.1109/ICASIC.2007.4415567.

[26] Schwarz, Eric. (2007). Binary Floating-Point Unit Design. 10.1007/978-0-387-34047-0_8.

[27] H. Suzuki, H. Morinaka, H. Makino, Y. Nakase, K. Mashiko and T. Sumi, "Leading-zero anticipatory logic for high-speed floating point addition," in IEEE Journal of Solid-State Circuits, vol. 31, no. 8, pp. 1157-1164, Aug. 1996, doi: 10.1109/4.508263.

[28] M. S. Schmookler and K. J. Nowka, "Leading zero anticipation and detection-a comparison of methods," Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001, 2001, pp. 7-12, doi: 10.1109/ARITH.2001.930098.

[29] https://pages.cs.wisc.edu/~david/courses/cs552/S12/handouts/guardbits.pdf