DOI: 10.21917/ijme.2026.0396

DESIGN AND FPGA REALIZATION OF A HIGH-SPEED 32-BIT IEEE-754 FLOATING-POINT MULTIPLIER USING RADIX-4 BOOTH ENCODING AND DADDA TREE REDUCTION

Shyam Peraka, Pusuluri Vinod Babu, Ramineni Vardhan Naidu, Ravanam V.S.M. Sesha Suresh, Kuna Yashaswini and Shaik Gowsiya

Department of Electronics and Communication Engineering, Rajiv Gandhi University of Knowledge Technologies, India

Abstract

The IEEE-754 single-precision floating-point multiplication is a fundamental arithmetic operation in modern digital systems and is widely used in DSP, graphics and machine learning accelerators. The demand for computations is increasing day by day, so the hardware realization of these operations in terms of logic utilization and signal propagation delay is of utmost importance. This paper presents a highly optimized design of 32-bit floating point multiplier for FPGA. The design approach is purely structural with high emphasis on mantissa data-path. The proposed architecture uses a highly specialized Radix-4 Modified Booth Encoding algorithm to reduce the number of partial products by half. This leads to a substantial reduction of hardware overhead. We also propose a new narrow 25-bit Partial Product Generator and a dedicated wire-routed Two-Shift stage, which achieves around 50% gate reduction in the generation phase by eliminating redundant sign-extensions. A bounded Dadda reduction tree compresses the first 14-row matrix (13 partial products and one structural sign-correction row). The proposed tree is approximately 41% narrower than conventional 88-bit designs, thereby reducing internal interconnect and routing overhead. The last addition is done by 52-bit Carry Propagate Adder. The proposed architecture is synthesized on Xilinx Artix-7 platform and the results show the significant area saving and the deeply reduced latency, which makes the architecture very effective for high-performance computational applications with resource constraints.

Keywords:

IEEE 754, Floating-Point Multiplier, Radix-4 Modified Booth Encoding, Dadda Tree, FPGA Design

1. INTRODUCTION

Many areas involving heavy data processing, including Digital Signal Processing (DSP), 3D graphic rendering, scientific simulations, and Deep Learning (DL) Inference, rely on a certain streamlined approach. In these areas, the evaluation of floating-point arithmetic is crucial. Unlike fixed-point arithmetic, which has a constrained dynamic range, floating-point arithmetic can express both extremely large and extremely small values within the same data type. The floating-point standards have been set and popularized by the IEEE (Institute of Electrical and Electronics Engineers) with the IEEE 754 Standard. When dealing with the four fundamental arithmetic operations in floating-point representation (addition, subtraction, multiplication, and division), multiplication is said to be the most involved and resource demanding operation. Normally, the computation of multiplication in floating-point arithmetic is partitioned in three overlapping phases: computation of the output sign bit using an exclusive OR, addition of a biased exponent, and multiplication of the mantissas. The first two phases (determining the sign and the biased exponent) are relatively fast and simple. In contrast, the multiplication of mantissas dominates the total Area-Delay Product (ADP) of the floating-point computing unit.

For a single-precision IEEE-754 operand, the mantissa includes 24 bits (23 bits are explicit fractional bits and the remaining 1 bit is implicit). When multiplying two 24-bit operands, 24 partial products are generated, which are added to give a 48-bit unsigned product. Letting adders do this step in a straightforward manner is the simplest solution. However, array structures are suboptimal as they result in $\Theta(N)$ linear propagation delays due to carry ripple cascades across the half and full adders. To alleviate the bottleneck that this latency causes, methods such as Wallace and Dadda tree multipliers have been used.

The Dadda tree is efficient in the number of adders used because it only compresses a row if the row exceeds the stage height limit, and the row is allowed to be the maximum height otherwise. However, even with such a method there would still be a great number of partial products, and as a result, a huge amount of hardware. To solve this issue, Modified Booth Encoding (MBE) Radix-4 would be used, and this would reduce the number of partial products by about half. The combination of Radix-4 Modified Booth Encoding and Dadda tree reduction effectively reduces the number of partial products and the associated reduction complexity. Building upon these techniques, the proposed architecture further optimizes partial product generation through a narrow 25-bit Partial Product Generator and a dedicated two-shift stage, reducing redundant sign extensions and minimizing the internal datapath width.

This paper provides a new and structurally constrained architecture that addresses the above inadequacies. In addition, the structural alignment and reduction process is limited to a 52-bit constrained Dadda tree, leading to approximately 41% reduction in bit width as opposed to industry-standard 88-bit tree processes.

2. LITERATURE SURVEY

Floating point multiplication is one of the essential requirements in today's VLSI and FPGA-based technology design as it strives to maximize efficiency in terms of hardware utilization, speed, and power consumption. The following part will highlight the motivation of the proposed approach by examining three recent scientific works in the area.

In [2], Sharma et al. proposed an advanced FPGA multiplier architecture that extensively made use of Carry-Lookahead Adders (CLAs) in the terminal stages of partial product matrix reduction. They were right in their assumption that the final CPA stage defines the critical path of the entire multiplier circuit. Through replacing regular Ripple Carry Adders (RCAs) with

CLAs, they managed to obtain significant latency reduction of 8.5% for the Wallace tree and 9.9% for the Dadda tree, respectively. At the same time, the speedup had a price – namely, high increase in the number of resources used due to the sophisticated carry-lookahead structure, which requires considerable numbers of discrete logic elements and Look-Up Tables (LUTs). Such a problem occurs in FPGA architectures with limited number of resources, making the area-speed compromise inevitable in such cases.

Krishnan et al. [3] presented a strictly pipelined single-precision floating point multiplier that made use of an extensively altered array structure. This architecture was intended to increase the clock frequency of the entire system by adding pipeline registers in-between the main combinatorial parts, thus splitting the long critical path in multipliers. Although they obtained a very good throughput and reduced the area locally as compared to the old-fashioned Booth multipliers, the architectural overhead is severe. The large-scale incorporation of pipelining stages resulted in huge overhead of D-Flip Flops (registers). This not only increased the area overhead but resulted in high dynamic power consumption because of continuous clocking of thousands of registers. Moreover, the deep pipeline led to an increase in absolute latency in terms of clock cycles, which made the design non-viable for use in the DSPs that require combinatorial single cycle datapath.

Furthermore, Gowreesrinivas and Samundiswary [4] have proposed an approach where a single precision floating point multiplier is designed using the Vedic multiplier of the sutra ‘Urdhva Tiryagbhyam’ instead of the standard mantissa multiplier along with several types of adders namely ripple carry, carry select, carry skip, and Sklansky’s parallel prefix adder for the mantissa reduction and exponent addition operations. The result from their simulation experiment carried out using the Xilinx ISE target platform indicated that the Vedic-Sklansky configuration had the least critical path delay time (23.92 ns). The Vedic-carry-skip combination yielded the lowest silicon area requirement (505 slices) but came at the expense of an elevated delay of 43.94 ns. It is evident from the above result that, in the case of Vedic multiplier design, the selection of adders alone leads to a compromise on the parameters of area and delay where there can be no optimization of both parameters together at once. Contrary to [4], which maintains a fixed partial product generation scheme in the Vedic multiplier architecture and alters only the adder circuit, the proposed multiplier architecture in this paper modifies the partial product generation stage using Radix-4 Modified Booth encoding and a limited 25-bit data bus width.

3. IEEE-754 FLOATING-POINT MATHEMATICAL FORMULATION

3.1 IEEE-754 SINGLE-PRECISION FLOATING-POINT FORMAT

The IEEE-754 standard defines the representation and arithmetic rules for binary floating-point numbers [1]. The single-precision (binary32) format allocates exactly 32 bits, partitioned into three specific contiguous fields: a 1-bit sign indicator (S), an 8-bit biased exponent (E), and a 23-bit fractional mantissa, also known as the significand (M). The mathematical value of a normalized floating-point number is expressed by the following equation:

$$Value = (-1)^{S} \times (1.M) \times 2^{E - Bias} \quad (1)$$

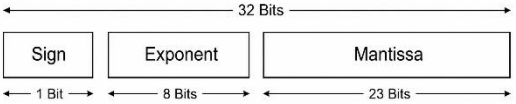


Fig.1. IEEE-754 Single Precision Floating Point Standard

For single-precision variables, the constant Bias is established as 127. The mantissa inherently possesses an implicit leading ‘1’, making the true functional width of the normalized significand 24 bits.

3.2 MULTIPLICATION OF TWO IEEE-754 SINGLE PRECISION FLOATING-POINT NUMBERS

The multiplication of two IEEE-754 floating-point operands, A and B, necessitates the parallel execution of three distinct mathematical operations. First, the Sign Evaluation determines the polarity of the resulting product. Because floating-point numbers utilize a sign-magnitude representation, the product’s sign is simply the exclusive-OR (XOR) of the input signs:

$$S_{out} = S_A \oplus S_B \quad (2)$$

Second, the Exponent Addition calculates the magnitude range. Since the exponents are biased by 127, a direct integer addition of EA and EB yields a double-biased result. Therefore, the singular bias must be subsequently subtracted to normalize the exponent domain:

$$E_{out} = E_A + E_B - 127 \quad (3)$$

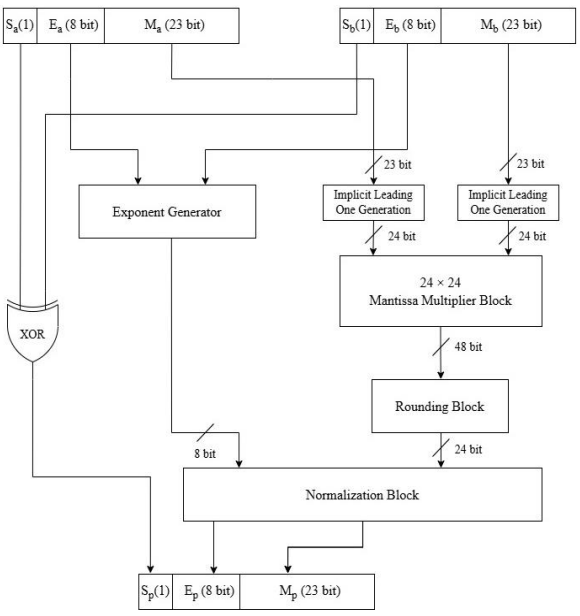


Fig.2. Overall Architecture of the Single Precision Floating Point Multiplier

Third, and most critically, is the Mantissa Multiplication. The 24-bit unsigned mantissas (comprising the implicit ‘1’ concatenated with the 23-bit fractional ‘*M*’) must be multiplied

together. This unsigned integer multiplication produces a 48-bit product (*P*).

$$P_{48bit} = (1.M_A) \times (1.M_B) \quad (3)$$

Finally, the system must perform Normalization and Rounding. The 48-bit product takes the format of XX.YYYY... If the value of the MSB, which is bit 47, is '1', then the result is normalized by shifting the radix point one place to the left and increasing the value of Eout by one. If bit 47 is '0', the product is already normalized. The terminal 23 bits of the output mantissa are then extracted, typically utilizing Round-to-Nearest-Even logic, finalizing the floating-point output.

3.3 INTRODUCTION TO BOOTH MULTIPLIER

The concept of hardware multiplication has been pioneered by Andrew D. Booth in the year 1951 [10]. This is an innovative signed binary multiplication approach used to overcome the drawback of conventional shift and add scheme of multiplication process. The main idea behind this is that a sequence of consecutive 1s in the binary multiplier can be mathematically compressed where a run starting at bit location i and ending at bit location j can be represented as "2j+1 – 2i". This conversion led to the representation of the multiplier in signed digit format (+1, -1, 0) such that zeros did not add any partial product.

The original Booth's algorithm based on the radix-2 system had an inherent limitation that, for alternating patterns of the multiplier such as 10101010, there would be no reduction in the number of partial products; hence, the worst-case complexity would become the same as ordinary multiplication. This prompted MacSorley in 1961 [11] to take Booth's idea and modify it to radix-4 and come up with what is known as Modified Booth's Encoding (MBE) today. MBE works by examining overlapping groups of three bits each in the multiplier, i.e., the current bit X2i+1, the next bit X2i, and the previously examined bit X2i-1. In doing so, MBE looks at two multiplier bits for each encoding, whereas Booth looked at one. The overlap of adjacent groups guarantees the examination of all bits in the multiplier and not any single bit at a time; the three-bit grouping results in five signed multiples of the multiplicand i.e., {0, +a, -a, +2a, -2a}.

The simplicity of the mathematical operations required for the five-element set described herein is precisely the driving force behind the appeal of its hardware realization. Multiples of +a and -a are easily inferred from the multiplicand and its two's complement correspondingly, while +2a and -2a can trivially be obtained using a single left shift. Odd multiples, like 3a, which would require expensive pre-addition, have been omitted from the adopted encoding scheme.

In the case of the k-bit operand used for multiplication, the scheme with three bits overlapping guarantees that no matter what the value of the operand is, there will always be ⌈k/2⌉ partial products to work with.

In the case of the design under consideration, the 24-bit mantissa operand is unsigned, whereas Modified Booth encoding requires a signed operand; the operand is, therefore, extended with zeroes prior to grouping, resulting in 13 overlapping Booth groups instead of 12 (Section 4.2). It nevertheless still decreases the partial product terms from 24 down to 13, representing a 46% reduction when compared to a traditional array multiplier. A correction row is then added to the data path in order to account for the two's complement property implied by the Booth encoded representation, resulting in a total of 14 rows before the Dadda reduction tree stage.

4. PROPOSED HARDWARE ARCHITECTURE

In the computational chain described above, the main source of the performance bottleneck is undoubtedly the 24×24-bit mantissa multiplier. The unique feature of the proposed hardware architecture is the redesign of the 24×24-bit mantissa multiplier to employ a dedicated structural dataflow rather than relying on synthesis tools to infer the multiplier from behavioral arithmetic operators such as (*). In particular, this dataflow is tailored to be used on modern FPGAs, namely the Xilinx Artix-7 series.

4.1 MANTISSA MULTIPLIER OVERVIEW

The architecture is essentially separated into five pure combinatorial stages: Radix-4 Booth Encoding, Partial Product Formation, Two-shift and Structural Alignment, Bounded Dadda Tree Reduction, and Final Carry Propagate Addition. The combination of all these stages ensures the avoidance of data-width explosion along with minimizing the internal signal routing. The Fig.3 shows the block diagram of the proposed mantissa multiplier.

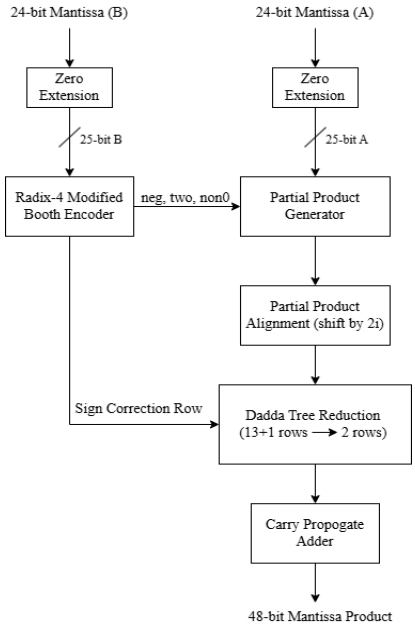


Fig.3. Block Diagram of Proposed 24x24 bit Mantissa Multiplier

4.2 RADIX-4 MODIFIED BOOTH ENCODER

Traditional array multipliers generate N partial products for an N×N multiplication (i.e., 24 rows for the mantissa). Radix-4 Modified Booth Encoding (MBE) mathematically halves this requirement by inspecting the multiplier operand (B) in overlapping groups of three bits (B[2i+1], B[2i], B[2i-1]) and encoding them into a set of command signals that multiply the multiplicand (A) by a factor from the set {-2, -1, 0, +1, +2}.

Because the mantissas are unsigned 24-bit integers, the multiplier B is zero-extended to 25 bits, and subsequently extended to 27 bits (Bext) to ensure proper grouping and sign

handling. This generates exactly 13 overlapping 3-bit Booth groups. In standard MBE literature, the encoder translates these 3-bit groups into three distinct control signals: ‘neg’ (for negative factors), ‘two’ (for magnitude 2 factors), and ‘zero’ (for null factors). The proposed architecture optimizes this Boolean logic by condensing the ‘one’ and ‘zero’ signals into a single, highly efficient ‘non0’ control signal.

x_{i+1}	x_i	x_{i-1}	y_{i+1}	y_i	$z_{i/2}$	neg	two	non0
0	0	0	0	0	0	0	0	0
0	0	1	0	1	1	0	0	1
0	1	0	1	−1	1	0	0	1
0	1	1	1	0	2	0	1	1
1	0	0	−1	0	−2	1	1	1
1	0	1	−1	1	−1	1	0	1
1	1	0	0	−1	−1	1	0	1
1	1	1	0	0	0	0	0	0

Fig.4. Modified Radix-4 Booth Encoding Table and Respective Control Signals

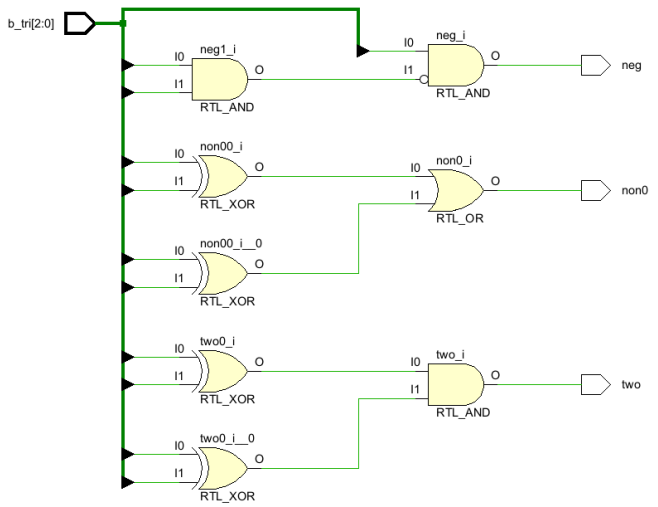


Fig.5. RTL Schematic of the Radix-4 Modified Booth Encoder.

The Karnaugh Map derivations for these optimized signals are defined by the following Boolean equations:

$$neg = B_{ext}[2i+1] \,\&\, \sim (B_{ext}[2i] \,\&\, B_{ext}[2i-1]) \quad (5)$$

$$two = (B_{ext}[2i+1] \oplus B_{ext}[2i])$$

$$(B_{ext}[2i+1] \oplus B_{ext}[2i-1]) \quad (6)$$

$$non0 = \left(B_{ext}[2i+1] \oplus B_{ext}[2i]\right) |$$

$$(B_{ext}[2i] \oplus B_{ext}[2i-1]) \quad (7)$$

By instantiating 13 independent Booth Encoders, the synthesizer maps these equations directly into single 3-input LUTs on the Artix-7 fabric, avoiding the inference of slower ROM-based lookup tables and bypassing the need to explicitly route a ‘zero’ flag to the subsequent multiplexers.

4.3 PARTIAL PRODUCT GENERATOR (PPG)

The job of the Partial Product Generator (PPG) is to multiply A by factors such as {-2, -1, 0, +1, +2}. Traditional Booth algorithms have a PPG that generates 2A internally by shifting A to the left, and computes (-A), (-2A) by finding their two’s complement. These partial products are generally extended to the size needed by the reduction tree.

By contrast, the Narrow PPG approach segregates the process of producing the partial products from the process of their positioning and works only on the 25-bit data path. This PPG circuit is regulated by the non0, neg, and two signals from the Booth encoder. The partial product is created using a simple 2:1 multiplexer along with the selective complementation. In case non0 signal is active, the two signal chooses between A and 2A signals whereas the neg determines whether selective complementation should be performed to produce either positive or negative partial product.

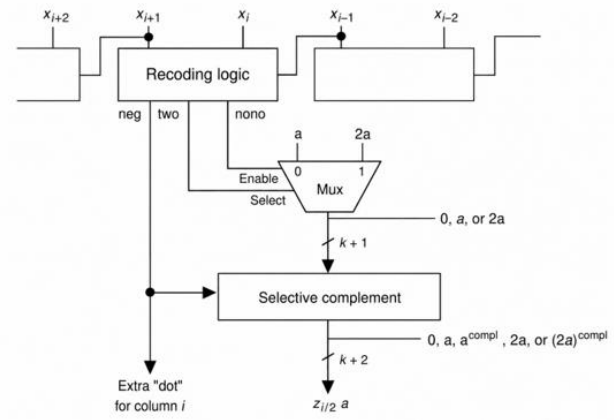


Fig.6. Modified Radix-4 Booth Multiplication Algorithm Architecture.

Each of the 13 PPG units works on a 25-bit datapath only and uses a smaller 25-bit 2:1 multiplexer rather than using a large multiplexer at the end accumulation width. The use of this narrower-width design results in reducing the logic and routing cost of the partial product generation block.

After the Narrow PPG, in the Two-Shift stage, each 25-bit raw partial product is expanded into a 26-bit partial product through hardwiring concatenation without the use of a barrel shifter. When two = 0 (±A), there is sign extension, while when two = 1 (±2A), there is shifting of the vector by 1 bit along with inserting of neg to the LSB, and this gives the necessary 1’s complement for −2A:

ppwide = {ppraw, encneg} (When two=1) (8)

These 26-bit partial products are now sign extended and structurally aligned in a 50-bit bus based on their weights in radix-4, with shifting of 2 bits per row. There is a maximum shift of 24 bits, and hence the maximum width is 26+24 = 50 bits. There is also a separate 50-bit row for sign correction (+1 to negative partial products), giving a total of 14 rows (13 partial products and 1 correction row) for Dadda reduction.

4.4 BOUNDED DADDA REDUCTION TREE

With the 14 rows properly aligned within a constrained 50-bit framework (padded with 2 extra guard bits to a safe 52-bit tree

width to absorb cascaded carry growth), the vectors enter the Dadda Reduction Tree.

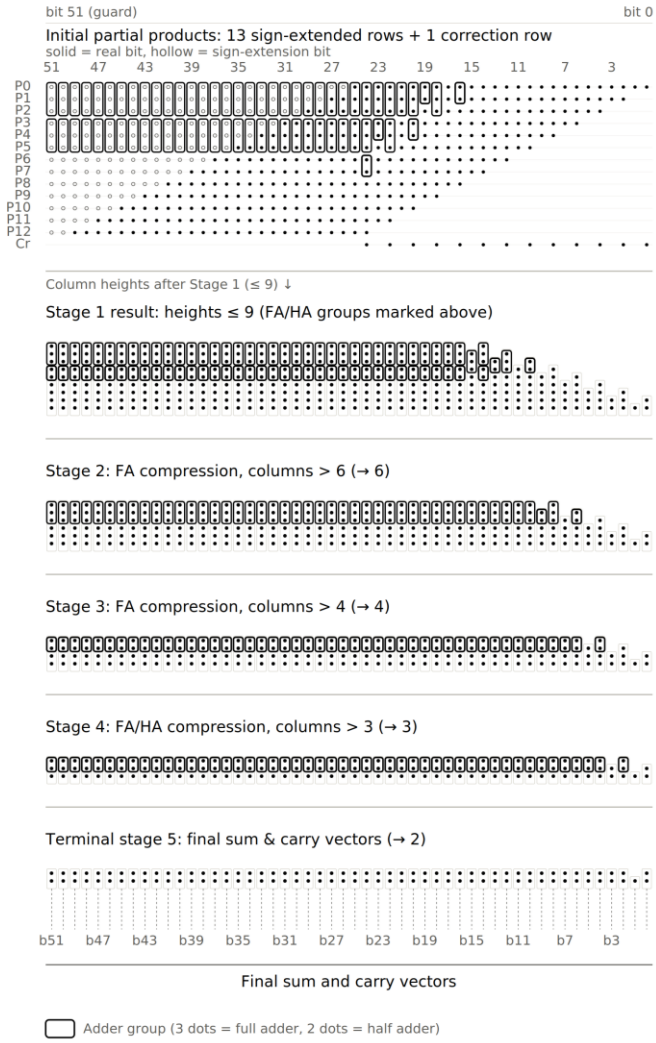


Fig.7. Dadda Tree Dot Diagram Illustrating the Five-Stage Compression of 14 Partial Product Rows down to 2 Terminal Rows.

Dadda trees operate on the principle of minimizing the number of Full Adders (FAs) and Half Adders (HAs) by only reducing column heights when they exceed the theoretical maximum allowed height for a given stage, determined by the sequence:

$$dj = floor(1.5 * dj - 1),\ d_0 = 2 \tag{9}$$

For an initial matrix possessing 14 rows, the Dadda sequence dictates exactly five compression stages to reach the terminal 2 rows. The height reduction strictly follows the numerical progression: 14 → 9 → 6 → 4 → 3 → 2. The Fig.7 visually depicts the structural dot-diagram of this rigorous reduction process.

The first stage sees compression for columns greater than 9 bits in height by using FAs, transmitting the sum bit through the current column, and the carry bit to the next left column. As a result, the height of the matrix will reduce to 9 bits at most. In the second stage, a very aggressive approach to columns greater than 6 bits in height is applied, using many FAs and compressing the matrix down to 6 bits in height. In this process of cascading, the heights of 4 bits and 3 bits are achieved successively, with the matrix being completely compressed to 2 bits height in the fifth stage.

4.5 TERMINAL CARRY PROPAGATE ADDER (CPA)

The ultimate stage of the mantissa multiplier necessitates the mathematical assimilation of the terminal Sum and Carry vectors produced by the Dadda tree. This is executed by a standard Carry Propagate Adder. In the proposed design, a 52-bit Ripple Carry Adder (RCA) is instantiated.

While RCAs exhibit a theoretical Θ(N) linear latency due to the carry chain propagating from the Least Significant Bit (LSB) to the Most Significant Bit (MSB), the implementation capitalizes on the highly optimized, dedicated fast-carry logic circuitry embedded within the Xilinx Artix-7 logic slices (the CARRY4 primitives). Consequently, the propagation delay across the RCA is sufficiently minimized, negating the requirement to deploy area-hungry Carry-Lookahead Adders as suggested by earlier literature.

5. FPGA SYNTHESIS AND TIMING RESULTS

To rigorously evaluate the performance metrics of the proposed floating-point multiplier architecture, the complete design was modeled using IEEE-compliant Verilog-2005 structural syntax. Logic synthesis, placement, routing, and comprehensive timing analysis were executed targeting the Xilinx Artix-7 FPGA platform (device: xc7a100tcsg324-1) utilizing the Vivado Design Suite.

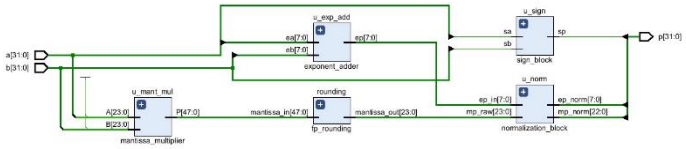


Fig.8. RTL Schematic of Proposed Architecture

5.1 SIMULATION AND FUNCTIONAL VERIFICATION

Prior to physical synthesis, exhaustive functional verification was performed using Xilinx Vivado 2023.2. Custom testbenches were engineered to subject both the isolated 24×24-bit mantissa multiplier and the overarching floating-point unit to thousands of randomized and edge-case test vectors. The vectors rigorously validated computational accuracy against IEEE-754 standards, specifically targeting complex scenarios including Not-a-Number (NaN) propagation, infinite boundaries, subnormal operand handling, and severe rounding extremities.

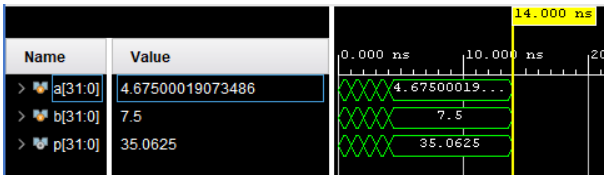


Fig.9. Comprehensive Functional Simulation Waveforms Validating Radix-4 Encoding and Sub-module Propagation.

5.2 COMPARATIVE ARCHITECTURAL ANALYSIS

From the results of the syntheses, one can note the efficiency of structural optimizations suggested for the datapath. As seen from the results above, elimination of redundant sign extensions and split of the shift operation into the Narrow Partial Product Generator leads to 50% savings of the number of logic cells needed for the implementation of the partial product generation stage of the architecture. This savings is mostly provided by restricting the datapath of the PPG to 25 bits and usage of the compact 2:1 multiplexers in place of wide multiplexing structures. Also, restriction of the Dadda reduction tree and Carry Propagate Adder to the necessary 52 bits datapath leads to 41% reduction in the number of hardware utilized compared to the classical array implementation. Reduction of the datapath width saves unneeded logic and routing resources, but does not affect the required arithmetic precision of the architecture. From the results above, it is clear that the suggested combination of narrow partial product generation, structural alignment and limited tree reduction provides an efficient area-efficient architecture without any additional computing stages. Table I shows a synthesized comparison of the suggested architecture with other benchmark architectures from the literature.

The success of the above architecture is proven empirically in Table.1 via the absolute superiority of the above architecture over Sharma et al. [2] due to the clever design of not requiring large space usage through Carry-Lookahead Adders and a smaller Dadda tree to overcome the latency issue of RCAs. Furthermore, the above architecture manages to avoid the problem of register explosion as well as high power requirements by the pipelined arrays in [3].

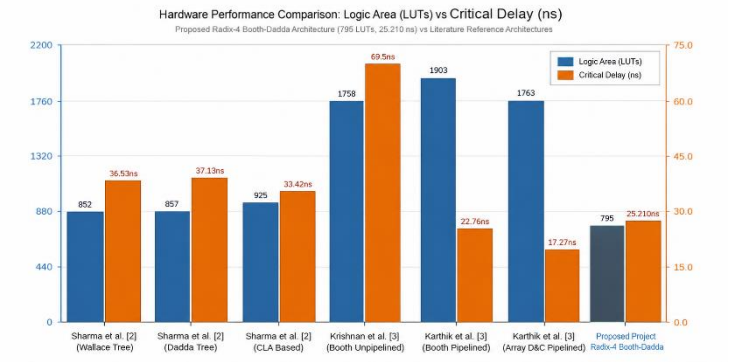


Fig.10. Graphical Comparison Illustrating Area Utilization (LUTs) and Signal Latency (ns)

Table.1. Comparative Analysis of State-of-the-Art Multiplier Architectures

Architecture Reference	Logic Area (LUTs)	Critical Delay (ns)	Target Platform
Sharma et al. [2] (Wallace Tree)	852	36.53	FPGA
Sharma et al. [2] (Dadda Tree)	857	37.13	FPGA
Sharma et al. [2] (CLA Based)	925	33.42	FPGA
Krishnan et al. [3] (Booth Pipelined)	1903	22.763	FPGA
Krishnan et al. [3] (Array D & C Pipelined)	1763	17.266	FPGA
Proposed (Booth+Dadda Tree+CPA)	795	25.210	FPGA (Artix-7)

6. CONCLUSION AND FUTURE SCOPE

In this paper, a comprehensive description of the design, mathematical formulation, and hardware implementation of an extremely optimized IEEE-754 single-precision floating-point multiplier has been presented. The major novelty associated with this work is the aggressive optimization of the data-path for the mantissa calculation. By coming up with an extremely narrow Partial Product Generator, and entirely decoupling the left shift operations to a zero-logic, wire-routed concatenation phase, the internal data width of the Radix-4 Booth encoding has been minimized drastically. The number of rows for partial product is reduced from 24 to 14 using the Radix-4 Modified Booth algorithm, which operates on the bits of the multiplier in overlapping 3-bit sections, and creates corresponding partial products. The number of rows is substantially reduced entering the Dadda tree after this step. The above-mentioned hardware-oriented approach provided a remarkable logic gate reduction of about 50% only within the generation phase. Additionally, the succeeding Dadda reduction tree and ultimate Carry Propagate Adder have been limited to a maximum width of 52 bits, which is 41% narrower than traditional implementations. Synthesis comparisons from current literature provide solid proof that the proposed architecture provides an excellent balance, enabling high-speed single cycle combinational arithmetic with minimal hardware area.

Possible future research directions include the extension of the structural datapath optimization algorithms to the implementation of double-precision and half-precision IEEE-754 floating point number representation (i.e., 64-bit and 16-bit numbers). Another research direction includes the use of parameterized ultra-shallow pipelining designs to facilitate the high throughput DSP and parallel computing applications. Moreover, the reimplementation of the final Carry Propagate Adder using faster adder designs (e.g., Kogge-Stone and others) may be considered to further decrease the critical path delay.

REFERENCES

[1] IEEE 754-2008, “IEEE Standard for Floating-Point Arithmetic”, Available at: https://standards.ieee.org/ieee/754/4211/, Accessed in 2008.

[2] Abhay Sharma, M. Irfanul Hasan and Devesh Tiwari, “FPGA Implementation of Floating-Point Multiplier Employing Carry-Lookahead Adders”, *Proceedings of IEEE International Conference on Device Intelligence, Computing and Communication Technology*, pp. 1-7, 2025.

[3] T. Krishnan and S. Saravanan, “Design of Low-Area and High-Speed Pipelined Single Precision Floating Point Multiplier”, *Proceedings of IEEE International Conference on Advanced Computing and Communication Systems*, pp. 34-42, 2020.

[4] K.V. Gowreesrinivas and P. Samundiswary, “Comparative Study on Performance of Single Precision Floating Point

Multiplier using Vedic Multiplier and different types of Adders", *Proceedings of IEEE International Conference on Control, Instrumentation, Communication and Computational Technologies*, pp. 466-471, 2016.

[5] X. Jiang, "Performance Effects of Pipeline Architecture on an FPGA-Based Binary32 Floating Point Multiplier", *Microprocessors and Microsystems*, Vol. 34, No. 2, pp. 1-17, 2013.

[6] C.S. Wallace, "A Suggestion for a Fast Multiplier", *IEEE Transactions on Electronic Computers*, Vol. 13, No. 1, pp. 14-17, 1964.

[7] L. Dadda, "Some Schemes for Parallel Multipliers", *Alta Frequenza*, Vol. 34, pp. 349-356, 1965.

[8] N. Bai, "Logic Design and Power Optimization of Floating-Point Multipliers", *IEEE Access*, Vol. 13, pp. 1-26, 2022.

[9] L. Kavana and G.N. Keshava Murthy, "Power and Area-Efficient Multiplier Architectures: A Comparative Study of Array, Dadda, Booth, Wallace Tree, and Vedic Multipliers", *Proceedings of IEEE International Conference on Smart Systems for Applications in Electrical Sciences*, pp. 1-12, 2025.

[10] A.D. Booth, "A Signed Binary Multiplication Technique", *Quarterly Journal of Mechanics and Applied Mathematics*, Vol. 4, No. 2, pp. 236-240, 1951.

[11] O.L. MacSorley, "High-Speed Arithmetic in Binary Computers", *Proceedings of the IRE*, Vol. 49, No. 1, pp. 67-91, 1961.