DOI: 30.0485/ijearst.10.01.3495 https://ijearst.co.in/

ENGINEERING IN ADVANCED RESEARCH SCIENCE AND TECHNOLOGY

ISSN 2386-7876
Vol.02, Issue.02
July-2026
Pages: 1122-1134

High-Speed Approximate Hybrid Floating-Point Multiplication Using Radix-16 Modified Booth Encoding for Butter fly unit

[1]CH.MANASA SATYA KUMARI, [2]G.V. RAMANA
[1]M. Tech, Dept. of ECE, V S M College of Engineering, Ramachandrapuram, A.P, India.
[2]Assistant Professor, Dept. of ECE, V S M College of Engineering, Ramachandrapuram, A.P, India

ABSTRACT: This project presents floating-point operations and applies them to the implementation of Butterfly unit. The fused operations are a two-term dot product and an add-subtract unit. Floating-point (FP) multipliers are critical components in high-performance computing systems, particularly in signal processing, graphics, and machine learning applications. However, traditional FP multipliers face challenges in power, area, and latency efficiency. This paper proposes two Approximate Hybrid Floating Point Multipliers (AHFPM1 and AHFPM2), which integrate hybrid Radix-4 and Radix-8 Booth encoding techniques to improve computational efficiency while maintaining high accuracy. AHFPM1 utilizes an approximate 3Y recoding adder to accelerate partial product generation, whereas AHFPM2 employs a novel approximate encoder (AHFPE2) derived through Karnaugh map simplifications to reduce hardware complexity. Both designs selectively apply exact and approximate logic to different sections of the mantissa, enabling precision-aware optimization. Further, this project is enhanced by using Dynamic configurable Carry save adder based Radix16 booth encoding multiplier to further reduce power, density and time.

Keywords: approximate encoder, Floating point, Fourier transform, Radix, Booth encoding, Carry save adder, Normalization.

INTRODUCTION: The floating-point number system is attractive for various applications such as graphics, signal processing, and scientific calculations because it can provide an enormous dynamic range. This large range of the floating-point number system enables designers to overcome problems such as overflow that often occur in fixed point implementation. Among the various floating-point number formats that are used, the IEEE-754 standard is used in this report. The IEEE-754 floating point standard is frequently used in Intel, Apple, and most UNIX systems. Floating-Point arithmetic provides a wide dynamic range, freeing special purpose processor designers from the scaling and overflow/underflow concerns that arise with fixed-point arithmetic. Use of the IEEE-754 standard 32-bit floating-point format

also facilitates using the fast Fourier transform (FFT) processors as coprocessors in collaboration with general purpose processors. This paper is concerned with the efficient floating-point implementation of the butterfly units that perform the computations in FFT processors. Since many signal processing applications need high throughput more than low latency, the primary focus of the fused elements is to reduce the circuit area with secondary attention to reducing the delay. Two fused floating-point primitive operations have been developed recently to reduce the delay and area of FFT computation units. The first of the fused operations is a fused floating-point two-term dot product unit (Fused DP). The Fused DP is an extension of the fused multiply-add (FMA) operation that was developed initially for the IBM RS/-6000 processor and recently added to IEEE Std-754. The floating-point FMA has several advantages over discrete floating-point adders and multipliers in a general purpose processor. The FMA reduces the latency of a multiplication followed by an addition. Also, a single FMA may be used to replace the floating-point adder and the floating-point multiplier in a system. Some DSP algorithms have been rewritten to take advantage of the presence of FMA units. For example, a radix-16 FFT algorithm that speeds up FFTs in systems with FMA units is described. The second of the fused operations is a fused add subtract unit (Fused AS). In FFT algorithms, both the sum and the difference of a pair of operands are needed frequently. In a fused implementation, the sum and the difference operations can share a substantial amount of operand alignment logic with the benefit of reducing the circuit area. In traditional implementations with discrete floating-point adders and multipliers, the dot product and add-subtract operations may be performed serially, which is relatively slow. Generally, they are performed in parallel, which is expensive both in silicon area and in power consumption, but which provides the best throughput.

LITERATURE SURVEY: 1. Swartzlander and Saleh (2012) – FFT Implementation with Fused Floating-Point Operations: Swartzlander and Saleh proposed fused floating-point operations for FFT processors. The FFT butterfly consists of multiple floating-point multiplications, additions, and subtractions. The authors introduced fused add-subtract and dot-product units to reduce the number of rounding stages and arithmetic operations. Their implementation achieved approximately 15% higher speed and 30% smaller area compared to conventional butterfly architectures. The work demonstrated that integrating floating-point arithmetic operations inside the butterfly structure can significantly improve FFT performance. This research established the importance of optimized floating-point multipliers in FFT butterfly units.

2. Kaivani and Ko (2015) – Area Efficient Floating-Point FFT Butterfly Architectures: Kaivani and Ko presented a floating-point FFT butterfly architecture based on multi-operand adders. The authors

observed that FFT butterflies contain several consecutive arithmetic operations that increase latency and hardware cost. By merging arithmetic operations using five-operand adders, they reduced hardware complexity and improved area efficiency. Simulation results showed that the proposed architecture was nearly 50% smaller than previous high-speed designs. Their work highlighted the importance of optimizing arithmetic blocks, particularly floating-point multipliers and adders, within FFT butterfly units.

3. Acharya and Beulet (2017) – Floating-Point FFT Butterfly Using Binary Signed Digit Multipliers

Acharya and Beulet proposed a floating-point FFT butterfly architecture employing Binary Signed Digit (BSD) multipliers and adders. FFT butterfly computation requires extensive multiplication by twiddle factors, making the multiplier a major contributor to delay and power consumption. The BSD multiplier reduced carry propagation and improved computational speed. Experimental results demonstrated enhanced performance for FFT operations while maintaining floating-point precision. The study emphasized that multiplier optimization directly impacts FFT throughput.

4. Kaivani et al. – Multi-Operand Adder Based Butterfly Units: Further work in floating-point FFT implementations focused on merging multiplication and addition operations inside butterfly structures. Researchers proposed architectures using three-, four-, and five-operand adders to reduce intermediate arithmetic stages. These designs minimized data movement and improved hardware utilization. The study concluded that efficient arithmetic units significantly reduce butterfly latency and area overhead.

EXISTING TECHNIQUE:

Floating point multiplication using Radix4 & Radix8 booth encoding for butterfly unit

Butterfly unit

Butterfly unit is actually a fused-multiply-add/sub (FMA) over complex operands. Fig. depicts a DIT butterfly unit which consists of a complex multiplier, a complex adder and a complex subtractor.

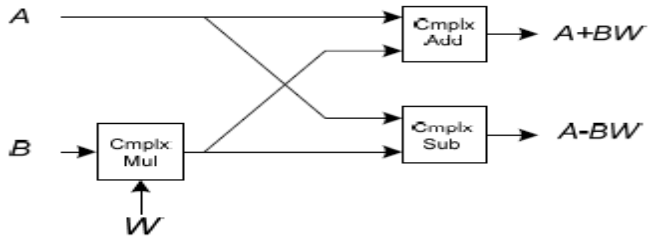

Fig 1: Butter fly unit

Butterfly architecture: The implementation of a DFT butterfly with expanded complex numbers using the conventional approach. Accordingly, it consists of four multipliers and six adders/subtractors. It should be noted that, given the constant values of twiddle factors (W), the multipliers are constant and can be implemented via a series of shifters and adders in lieu of the multiplier tree.

Floating-Point Fused Butterfly Arithmetic

This work [1] improves the performance of butterfly unit and hence FFT processors by proposing two fused floating-point operations, namely, Dot-Product and Add-Subtract The Dot-Product unit, compute AB+CD, consists of the following operations:

- Two floating-point multipliers, perform in parallel
- Alignment Shift
- (4:2) Compressor
- Carry-propagating adder, performs in parallel with Leading-Zero-Detector (LZD)
- Normalization & Rounding

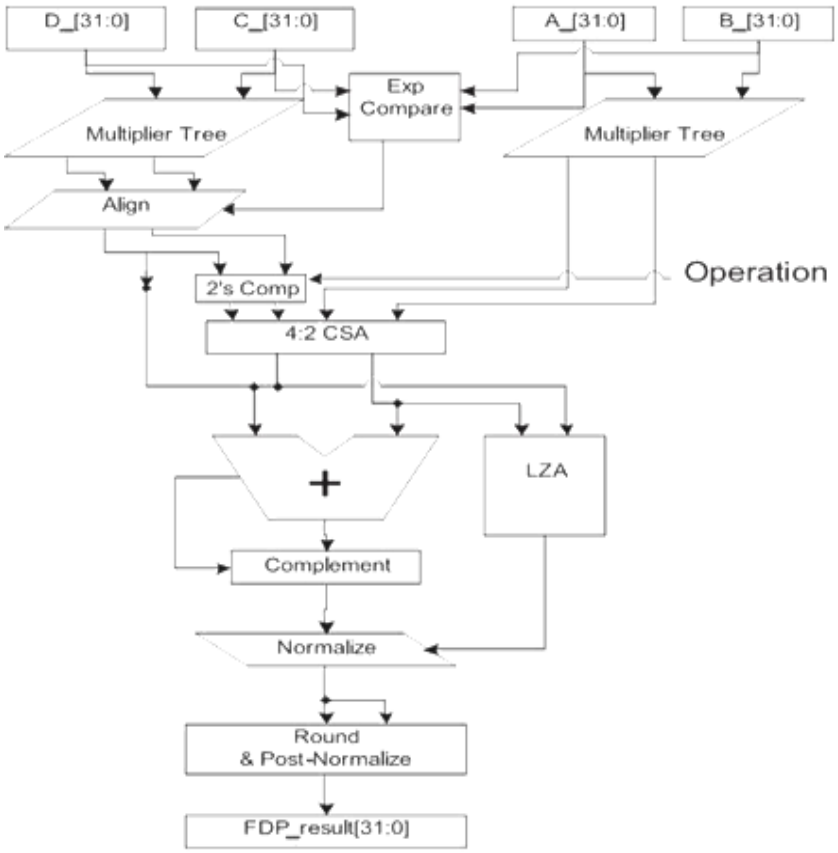

Fig 2: floating point multiplication

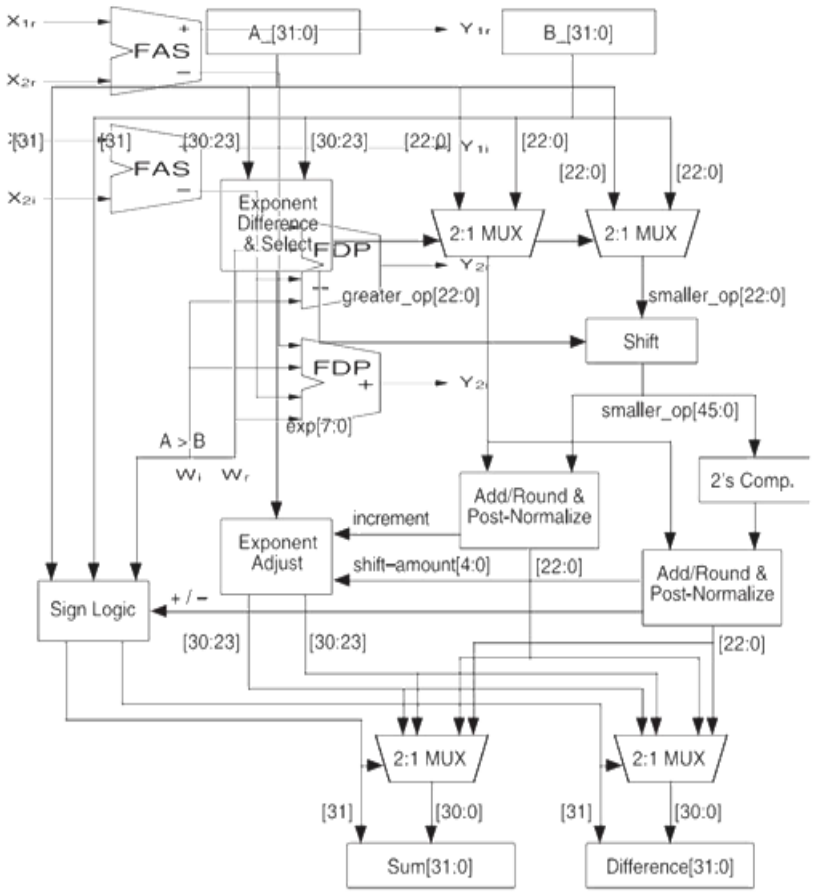

Fig 3: Floating pint add/sub unit

RADIX4 MODIFIED BOOTH: Modified Booth Algorithm Encoder: This modified booth multiplier is used to perform high-speed multiplications using modified booth algorithm. This modified booth multiplier's computation time and the logarithm of the word length of operands are proportional to each other. We can reduce half the number of partial product. Radix-4 booth algorithm used here increases the speed of multiplier and reduces the area of multiplier circuit. In this algorithm, every second column is taken and multiplied by 0 or +1 or +2 or -1 or -2 instead of multiplying with 0 or 1 after shifting and adding of every column of the booth multiplier. Thus, half of the partial product can be reduced using this booth algorithm. Based on the multiplier bits, the process of encoding the multiplicand is performed by radix-4 booth encoder. The overlapping is used for comparing three bits at a time. This grouping is started from least significant bit (LSB), in which only two bits of the booth multiplier are used by the first block and a zero is assumed as third bit as shown in the figure.

This shows the functional operation of the radix-4 booth encoder that consists of eight different types of states. The outcomes or multiplication of multiplicand with 0, -1, and -2 are consecutively obtained during these eight states.

Booth recoding table for radix-4

Multiplier Bits Block			Recoded 1-bit pair		2 bit booth	
i+1	i	i-1	i+1	i	Multiplier Value	Partial Product
0	0	0	0	0	0	Mx0
0	0	1	0	1	1	Mx1
0	1	0	1	-1	1	Mx1
0	1	0	1	0	2	Mx2
1	0	0	-1	0	-2	Mx-2
1	0	1	-1	1	-1	Mx-1
1	1	0	0	-1	-1	Mx-1
1	1	0	0	0	0	Mx0

Booth Recoding Table for Radix-4

The steps given below represent the radix-4 booth algorithm:

- Extend the sign bit 1 position if necessary to ensure that n is even.
- Append a 0 to the right of the least significant bit of the booth multiplier.
- According to the value of each vector, each partial product will be 0, +y, -y, +2y or -2y.

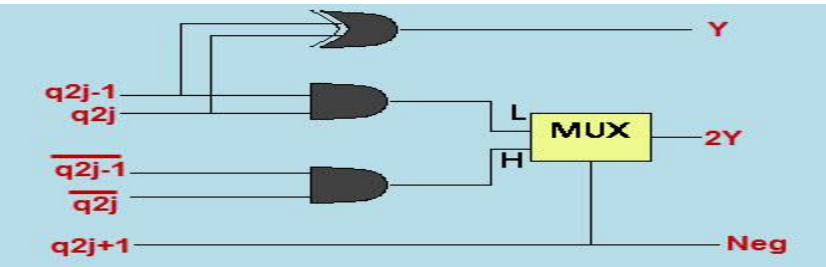

Fig 4: Booth's Encoder

Modified booth multiplier's (Z) digits can be defined with the following equation:

$Zj = q2j + q2j-1 - 2q2j+1$ with $q-1 = 0$

The figure shows the modified booth algorithm encoder circuit. Now, the product of any digit of Z with multiplicand Y may be -2y, -y, 0, y, 2y. But, by performing left shift operation at partial products generation stage, 2y may be generated. By taking 1′s complement to this 2y, negation is done, and then one is added in appropriate 4-2 compressor. One booth encoder shown in the figure generates three output signals by taking three consecutive bit inputs so as to represent all five possibilities -2X, -X, 0, X, 2X.

Partial Product Generator

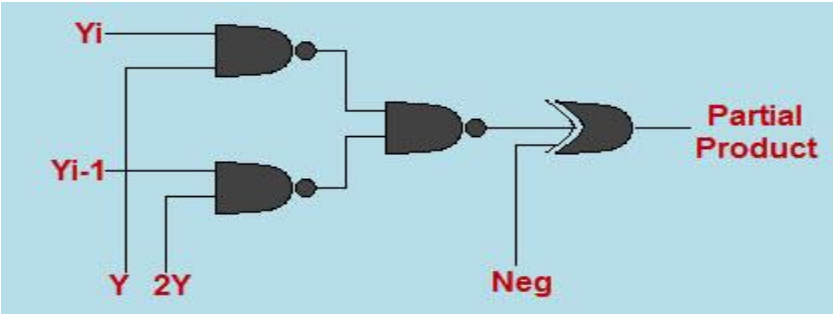

Fig 5: Partial Product Generator

If we take the partial product as -2y, -y, 0, y, 2y then, we have to modify the general partial product generator. Now, every partial product point consists of two inputs (consecutive bits) from multiplicand and, based on the requirement, the output will be generated and its complements also generated in case if required. The figure shows the partial product generator circuit.

RADIX-8 MODIFIED BOOTH ALGORITHM:

The Booth algorithm consists of repeatedly adding one of two predetermined values to a product P and then performing an arithmetic shift to the right on P.

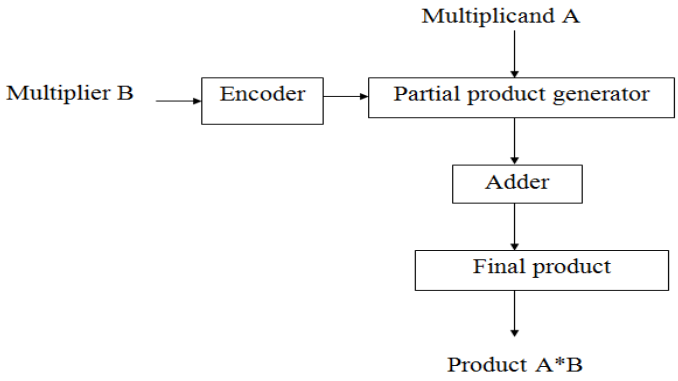

Fig. 6 Booth algorithm

The multiplier architecture consists of two architectures, i.e., Modified Booth. By the study of different multiplier architectures, we find that Modified Booth increases the speed because it reduces the partial products by half. Also, the delay in the multiplier can be reduced by using Wallace tree. The energy consumption of the Wallace Tree multiplier is also lower than the Booth and the array. The characteristics of the two multipliers can be combined to produce a high-speed and low-power multiplier. The modified

stand-alone multiplier consists of a modified recorder (MBR). MBR has two parts, i.e., Booth Encoder (BE) and Booth Selector (BS). The operation of BE is to decode the multiplier signal, and the output is used by BS to produce the partial product. Then, the partial products are added to the Wallace tree adders, similar to the carry-save-adder approach. The last transfer and sum output line are added by a carry look-ahead adder, the carry being stretched to the left by positioning.

Table . 2 Quartet coded signed-digit table

Quartet value	Signed-digit value
0000	0
0001	+1
0010	+1
0011	+2
0100	+2
0101	+3
0110	+3
0111	+4
1000	-4
1001	-3
1010	-3
1011	-2
1100	-2
1101	-1
1110	-1
1111	0

Here we have a multiplication multiplier, 3Y, which is not immediately available. To Generate it, we must run the previous addition operation: 2Y + Y = 3Y. But we are designing a multiplier for specific purposes and then the multiplier belongs to a set of previously known numbers stored in a memory chip. We have tried to take advantage of this fact, to relieve the radix-8 bottleneck, that is, 3Y generation. In this way, we try to obtain a better overall multiplication time or at least comparable to the time, we can obtain using a radix-4 architecture (with the added benefit of using fewer transistors). To generate 3Y with 21-bit words you just have to add 2Y + Y,ie add the number with the same number moved to a left position. A product formed by multiplying it with a multiplier digit when the multiplier has many digits. Partial products are calculated as intermediate steps in the calculation of larger products. The partial product generator is designed to produce the product multiplying by multiplying A by 0, 1, -1, 2, -2, -3, -4, 3, 4. Multiply by zero implies that the product is "0 ". Multiply by" 1 "means that the product remains the same as the multiplier. Multiply by "-1" means that the product is the complementary form of the number of two. Multiplying with "-2" is to move left one as this rest as per table.

PROPOSED METHOD: Floating point multiplication using configurable CSA based Radix16 booth encoding for butterfly unit.

RADIX-16 MODIFIED BOOTH ENCODING: Now days Analog systems are replacing by digital systems because of its high speed performance, takes less area and less power dissipation [1]. Multiplication is one of the most important and basic arithmetic operation that constitute programs. In fact 8.72% of all instructions in typical scientific programs are based on multiplication operation [2]. Many multipliers have been proposed in the past with consideration of small area, low power and high performance. Multiplication is achieved by the addition of a certain number of partial products rows. Each partial product row is generated by multiply the multiplier bit one by one to multiplicand. In a simple multiplier, the generated partial products rows are equal to the number of bits in multiplier. For example, in 8×8 bit multiplication,it will produce 8 partial product rows. It will take more adders and more time. To improve the performance of the multiplier, Booth multiplier is mostly used multiplier. The number of partial products rows that must be added to give the multiplication‟s result can be reduced by using Booth decoding. In Booth multiplier, the numbers of reduced partial products rows are depend on the grouping done at multiplier bits . These groups of multiplier perform the selected operation on multiplicand. In booth multiplier grouping is done by 2 bits, 3 bits, 4 bits and so on. Higher order booth decoding reduces the number of partial product rows by a greater by decoding larger groups of multiplier bits. This multiplication process is completed in 3 steps. First step: multiplier bits are divided in groups then these groups are fed to decoder at where it will indicate that which operation is to perform on multiplicand. Second step: here indicated operation performs on the multiplicand and it will generate the partial products. Third step: Now generated partial products are adding with adders.

RADIX16 MODIFIED BOOTH ENCODING ALGORITHM:

Booth recoding was originally introduced when multiplication was implemented using a series of shift-add operations. By recoding, the number of 1's in the multiplier could be reduced, and thereby the number of additions. The core idea is as follows:

Multiplier: $B=-bn-12n-1+\sum n-2i=0bi2i$

- Introduce new variables: $b\hat{}i=-bi+bi-1$ for $i=0\ldots n-1$ (assume $b-1=0$).
- Compute$P=\sum n-1i=0(b\hat{}i\times 2i\times A)$
- Although this looks very similar to the normal product, note that any time $bi=bi-1$ there is no addition to be performed as the partial product will be zero. In the case of serial addition, these steps can be skipped, thus saving computation.To reduce the number of partia l products added while multiplying the multiplicand higher radix Booth Encoding algorithm is one of the most well known techniques used .Radix 16 Booth algorithm which scan strings of five bits with the algorithm given below:

(1) Extend the sign bit 1 p osition if necessary to ensure that n is even.

(2) Append a 0 to the right of the LSB of the multiplier.

(3) According to the value of each vector,each Partial Product will be 0,y,+2y,+3y,+4y,+5y,+6y,+7y,+8y, -8Y, - 7y, - 6y, - 5y, - 4y, -3y, - 2Y, - Y. The multiplicati on of y is done by shifting y by one bit to the left. Thus, in any case, in designing n bit parallel multipliers, only n/4 partial products are generated

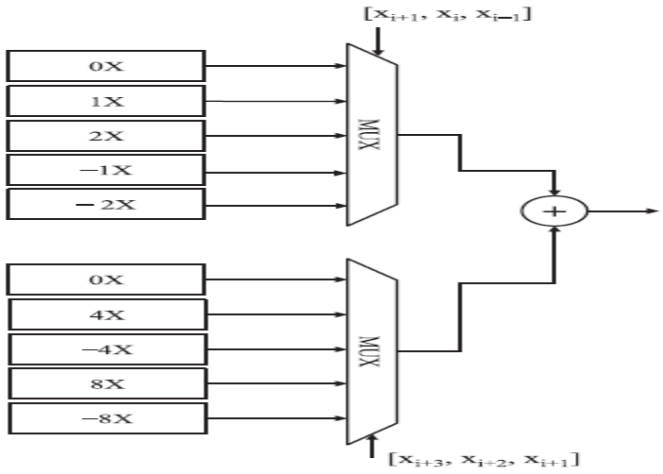

Fig 7: Radix16 modified booth encoder

X_{i+3}	X_{i+2}	X_{i+1}	X_i	X_{i-1}	PP
0	0	0	0	0	0Y
0	0	0	0	1	1Y
0	0	0	1	0	1Y
0	0	0	1	1	2Y
0	0	1	0	0	2Y
0	0	1	0	1	3Y
0	0	1	1	0	3Y
0	0	1	1	1	4Y
0	1	0	0	0	4Y
0	1	0	0	1	5Y
0	1	0	1	0	5Y
0	1	0	1	1	6Y
0	1	1	0	0	6Y
0	1	1	0	1	7Y
0	1	1	1	0	7Y
0	1	1	1	1	8Y
1	0	0	0	0	-8Y
1	0	0	0	1	-7Y
1	0	0	1	0	-7Y
1	0	0	1	1	-6Y
1	0	1	0	0	-6Y
1	0	1	0	1	-5Y
1	0	1	1	0	-5Y
1	0	1	1	1	-4Y
1	1	0	0	0	-4Y
1	1	0	0	1	-3Y
1	1	0	1	0	-3Y
1	1	0	1	1	-2Y
1	1	1	0	0	-2Y
1	1	1	0	1	-1Y
1	1	1	1	0	-1Y
1	1	1	1	1	0Y

Table 3: Radix16 modified booth encoding table

Carry Save Adder A carry-save adder or CSA is a type of digital adder mainly used for computing the sum of a minimum of three or above binary numbers very efficiently. A CSA is normally used within a binary multiplier because this multiplier involves the addition of the above two binary numbers after multiplication. By using this method, a big adder can be implemented which is very faster compared to the usual addition of numbers.

Carry Save Adder Block Diagram: The Carry Save adder circuit diagram is shown below. This type of adder is very different as compared to other types because it doesn't transmit the middle carries toward the next stages, but in its place, it saves the carry & addends to the sum of the next stage with another fuller adder (FA).

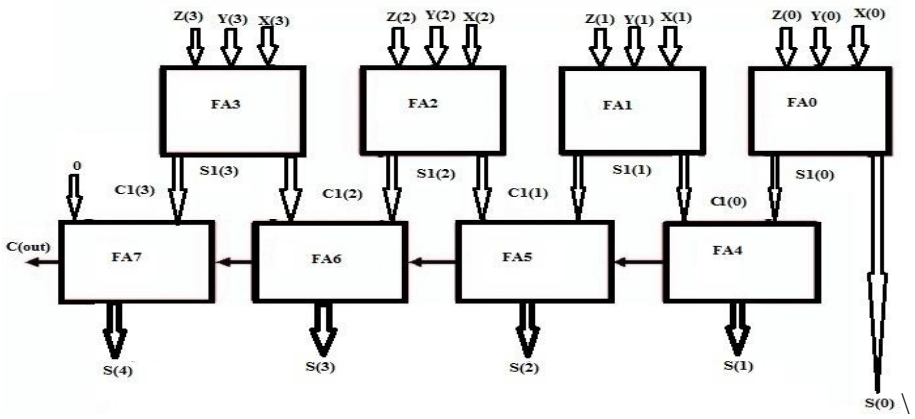

Fig 8: Carry save adder

The technique of adding up binary bits, the first stage of the addition part mainly includes saving the carries & sum bits, and transfers to the second stage. This stage acts related to ripple carry adder or RCA where the stored carry & sum bits are added. The operands used in this adder are three like X, Y & Z where 'Z' is a four-bit input carry. Here, for four every bit of X, Y & Z, and 4 FAs is used. For every FA, the sum & carry bits are produced. Here, the carry bits are not transmitted to the next FA, but in its place, they are simply saved & added up to the next sum term with a ripple carry adder.

Carry Save Adder Working: The carry save adder works by assembling K FAs without any horizontal connection. The main function of this adder is to add three k-bit integers like X, Y & Z to generate two integers sum 'S' & carry C. The carry propagator is propagated to the next level whereas the carry generator is used to generate the output carry, irrespective of the input carry. The carry propagation and generation are two functions in the carry save adder. The carry propagation (Cp) is propagated to the next level whereas the carry generator (Cg) is used to generate the output carry, irrespective of input carry.

RESULTS

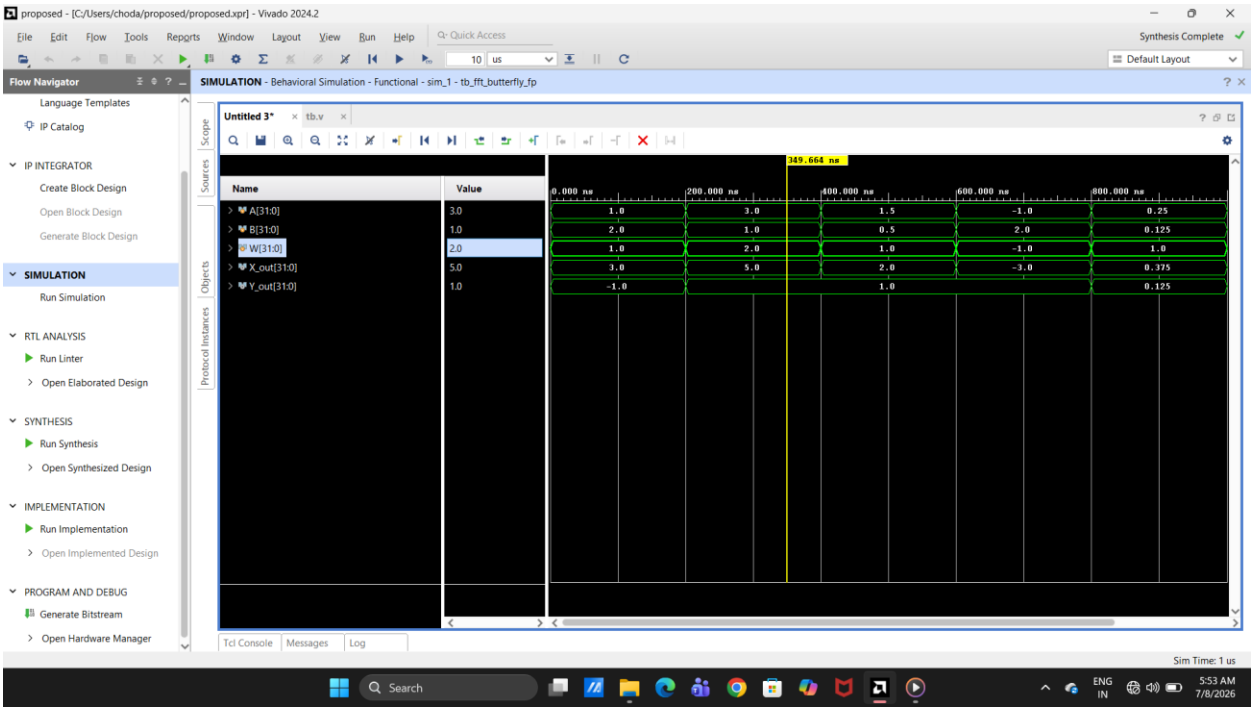

Fig: a Proposed simulation result

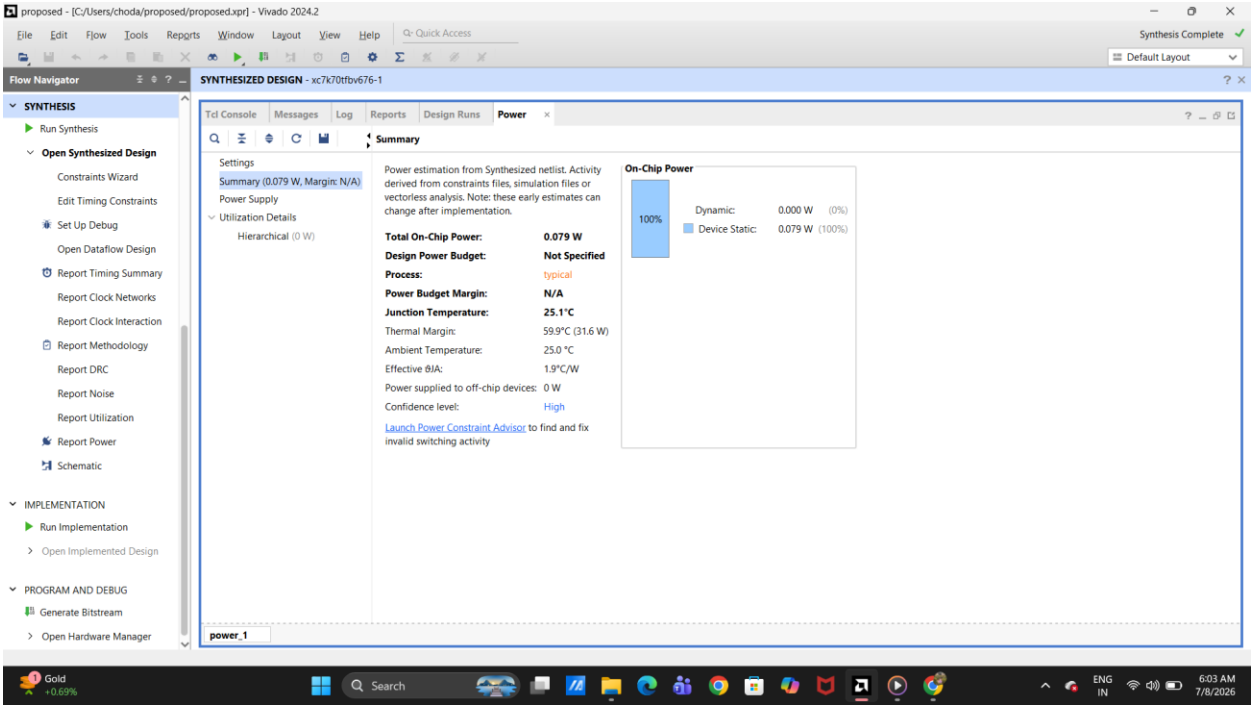

Fig:b Proposed Power report

Advantages

1. High-Speed Multiplication
    - Radix-16 Booth encoding significantly reduces the number of partial products compared to conventional Booth multipliers.
    - Fewer partial products lead to lower multiplication delay.
2. Reduced Hardware Complexity
    - Carry Save Adders (CSAs) efficiently combine multiple partial products without immediate carry propagation.
    - This reduces critical path delay and improves overall performance.
3. Lower Power Consumption
    - Approximate computing techniques reduce switching activity and logic utilization.
    - Suitable for power-constrained signal-processing systems.
4. Improved Area Efficiency
    - Reduced partial products and optimized adder structures decrease gate count and silicon area.

Applications

Digital Signal Processing (DSP)

- FIR and IIR filters
- Fast Fourier Transform (FFT)
- Discrete Cosine Transform (DCT)

Multimedia Processing

- Image enhancement
- Image compression (JPEG)
- Video encoding and decoding

Machine Learning and AI

- Neural network accelerators
- Deep learning inference engines
- Edge AI devices

Conclusion

The proposed approximate hybrid floating-point multiplier based on Radix-16 Booth encoding and Carry Save Adder architecture provides an efficient solution for high-performance signal-processing applications. The Radix-16 Booth algorithm reduces the number of generated partial products, while the CSA structure accelerates partial-product accumulation by eliminating intermediate carry propagation delays. Combined with approximation techniques, the design achieves significant improvements in speed, area, and power consumption while maintaining acceptable computational accuracy. Therefore, the proposed architecture is highly suitable for modern DSP, multimedia, and AI-based systems where performance and energy efficiency are critical requirements.

Future Scope

1. **Higher-Radix Multipliers**
    - Investigate Radix-32 or adaptive Booth encoding techniques for further reduction of partial products.

2. **Advanced Approximation Techniques**
    - Develop dynamically configurable approximation levels based on application requirements.
3. **FPGA and ASIC Optimization**
    - Implement the design on advanced technology nodes (7nm, 5nm, etc.) for improved performance.

References:

[1] E. E. Swartzlander Jr. and H. H. M. Saleh, "FFT implementation with fused floating-point operations," *IEEE Transactions on Computers*, vol. 61, no. 2, pp. 284–288, Feb. 2012, doi: 10.1109/TC.2010.271.

[2] A. Kaivani and S.-B. Ko, "Area efficient floating-point FFT butterfly architectures based on multi-operand adders," *Electronics Letters*, vol. 51, no. 12, pp. 895–897, Jun. 2015, doi: 10.1049/el.2015.0342.

[3] J. Sohn and E. E. Swartzlander Jr., "Improved architectures for floating-point fused dot-product unit," in *Proc. 21st IEEE Symposium on Computer Arithmetic (ARITH)*, Austin, TX, USA, 2013, pp. 38–41.

[4] J. H. Min, S. W. Kim, and E. E. Swartzlander Jr., "A floating-point fused FFT butterfly arithmetic unit with merged multiple-constant multipliers," in *Proc. 45th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, USA, 2011, pp. 520–524.

[5] A. F. Tenca, "Multi-operand floating-point addition," in *Proc. 19th IEEE Symposium on Computer Arithmetic (ARITH)*, Portland, OR, USA, 2009, pp. 161–168.

[6] P. Kornerup, "Correcting the normalization shift of redundant binary representations," *IEEE Transactions on Computers*, vol. 58, no. 10, pp. 1435–1439, Oct. 2009.

[7] IEEE Standards Association, *IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008)*, New York, NY, USA: IEEE, Aug. 2008.

[8] A. Kaivani and S.-B. Ko, "Floating-point butterfly architecture based on signed-digit representation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2015.

[9] S. Knowles, "A family of adders," in *Proc. 15th IEEE Symposium on Computer Arithmetic*, Vail, CO, USA, 2001, pp. 277–281.

[10] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. Boston, MA, USA: Addison-Wesley, 2011.

[11] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed. New York, NY, USA: Oxford University Press, 2010.

[12] M. Ercegovac and T. Lang, *Digital Arithmetic*. San Francisco, CA, USA: Morgan Kaufmann, 2004.

[13] A. Mukherjee, A. Sinha, and D. Choudhury, "A novel architecture of area efficient FFT algorithm for FPGA implementation," *arXiv preprint arXiv:1502.07055*, 2015.

[14] S. C. Hsu, S. J. Huang, S. G. Chen, S. C. Lin, and M. Garrido, "A 128-point multi-path SC FFT architecture," *arXiv preprint arXiv:2007.14736*, 2020.

[15] N. K. Unnikrishnan and K. K. Parhi, "Multi-channel FFT architectures designed via folding and interleaving," *arXiv preprint arXiv:2202.09623*, 2022.

[16] A. Mukherjee and D. Choudhury, "An area efficient 2D Fourier transform architecture for FPGA implementation," *arXiv preprint arXiv:1810.06885*, 2018.

[17] S. Knowles, "Carry-save arithmetic and high-speed multiplier design," *IEEE Computer Arithmetic Newsletter*, vol. 23, no. 2, pp. 12–18, 2002.

[18] O. L. MacSorley, "High-speed arithmetic in binary computers," *IRE Proceedings*, vol. 49, no. 1, pp. 67–91, Jan. 1961.

[19] A. D. Booth, "A signed binary multiplication technique," *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[20] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 1, pp. 14–17, Feb. 1964.