ISSN: 2582 - 6379
IJISEA Publications
International Journal for Interdisciplinary Sciences and Engineering Applications
IJISEA - An International Peer- Reviewed, Open Access Journal
2025, Volume 6 Issue 3
www.ijisea.org

FPGA-Based Implementation of IEEE 754 Single Precision Floating Point Multiplier

K. Vijaya Swathi[1*], P. Swarnalatha[2]
[1,2]Department of Electronics and Communication Engineering, Sri Venkateswara Institute of Science And Technology -Kadapa, India.
swathiprathap16@gmail.com[1]
DOI: 10.5281/zenodo.16728999

ABSTARCT

This paper presents an efficient implementation of a single-precision floating point multiplier compliant with the IEEE 754 standard, optimized for deployment on a Xilinx Virtex-5 FPGA. The design is developed using VHDL and follows a pipelined architecture to ensure high performance while maintaining technology independence. It includes mechanisms for handling overflow and underflow conditions; however, rounding is intentionally omitted to preserve higher precision, particularly for applications such as Multiply and Accumulate (MAC) operations. The implemented multiplier achieves a performance of 301 MFLOPs with a latency of three clock cycles. Functional verification was carried out against the Xilinx floating point IP core to ensure correctness.

Keywords: Floating Point, IEEE 754, Multiplier, FPGA, VHDL, Pipelining, MAC Unit

I. INTRODUCTION

Floating point representation is a widely used method for encoding real numbers in binary, and the IEEE 754 standard defines two primary formats: the Binary Interchange Format and the Decimal Interchange Format. Among these, multiplication of floating-point numbers plays a vital role in digital signal processing (DSP) applications that require a wide dynamic range. This paper specifically focuses on the single-precision normalized binary interchange format. As illustrated in Fig. 1, the IEEE 754 single-precision format consists of three components: a 1-bit sign (S), an 8-bit exponent (E), and a 23-bit fraction (M or mantissa). An implicit leading bit, '1', is added to the fraction to form the significand, resulting in a 24-bit value. A floating-point number is considered normalized when the exponent is greater than 0 and less than 255, and the most significant bit (MSB) of the significand is 1.

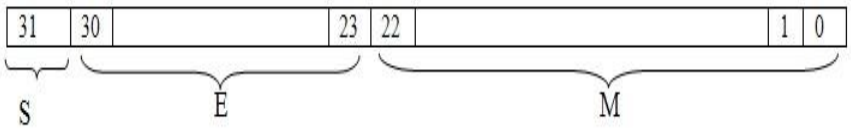


Figure1: Single precision IEEE floating point format

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

ISSN: 2582 - 6379
IJISEA Publications
International Journal for Interdisciplinary Sciences and Engineering Applications
IJISEA - An International Peer- Reviewed, Open Access Journal
2025, Volume 6 Issue 3
www.ijisea.org

Where M = m22 2-1 + m21 2-2 + m20 2-3+…+ m1 2-22+
m0 2-23; Bias = 127.
Significant is the mantissa with an extra MSB bit.
This research has been supported by Mentor Graphics.

Multiplying two numbers in floating point format is done by 1- adding the exponent of the two numbers then subtracting the bias from their result, 2- multiplying the significand of the two numbers, and 3- calculating the sign by XORing the sign of the two numbers. In order to represent the multiplication result as a normalized number there should be 1 in the MSB of the result (leading one). Floating-point implementation on FPGAs has been the interest of many researchers. In [2], an IEEE 754 single precision pipelined floating point multiplier was implemented on multiple FPGAs (4 Actel A1280). In [3], a custom 16/18-bit three stage pipelined floating point multiplier that doesn't support rounding modes was implemented. In [4], a single precision floating point multiplier that doesn't support rounding modes was implemented using a digit-serial multiplier: using the Altera FLEX 8000 it achieved 2.3 MFlops. In [5], a parameterizable floating-point multiplier was implemented using the software-like language Handel-C, using the Xilinx XCV1000 FPGA; five stages pipelined multiplier achieved 28MFlops. In [6], a latency optimized floating point unit using the primitives of Xilinx Virtex II FPGA was implemented with a latency of 4 clock cycles. The multiplier reached a maximum clock frequency of 100 MHz.

II. FLOATING POINT MULTIPLICATION ALGORITHM

As stated in the introduction, normalized floating point numbers have the form of

$$Z= (-1^{S}) * 2^{(E - Bias)} * (1.M).$$

To Multiply two floating point numbers the following is done:

1. Multiplying the significand; i.e. (1.M1*1.M2)
2. Placing the decimal point in the result
3. Adding the exponents; i.e. (E1 + E2 – *Bias*)
4. Obtaining the sign; i.e. s1 xor s2
5. Normalizing the result; i.e. obtaining 1 at the MSB of the results' significand
6. Rounding the result to fit in the available bits
7. Checking for underflow/overflow occurrence

Consider a floating-point representation similar to the IEEE 754 single precision floating point format, but with a reduced number of mantissa bits (only 4) while still retaining the hidden '1' bit for normalized numbers:

A = 0 10000100 0100 = 40, B = 1 10000001 1110 = -7.5
To multiply A and B
1. Multiply significand:

ISSN: 2582 - 6379
IJISEA Publications
International Journal for Interdisciplinary Sciences and Engineering Applications
IJISEA - An International Peer- Reviewed, Open Access Journal
2025, Volume 6 Issue 3
www.ijisea.org

1.0100
× 1.1110

00000 10100
10100
10100
10100
1001011000

2. Place the decimal point: 10.01011000
3. Add exponents: 10000100
+ 10000001
100000101

The exponent representing the two numbers is already shifted/biased by the bias value (127) and is not the true exponent; i.e. EA = EA-true + bias and EB = EB-true + bias And

EA + EB = EA-true + EB-true + 2 bias

So, we should subtract the bias from the resultant exponent otherwise the bias will be added twice.

100000101
- 01111111
10000110

4. Obtain the sign bit and put the result together:
1 10000110 10.01011000
5. Normalize the result so that there is a 1 just before the radix point (decimal point). Moving the radix point one place to the left increments the exponent by 1; moving one place to the right decrements the exponent by 1.
1 10000110 10.01011000 (before normalizing) 1 10000111 1.001011000 (normalized) The result is (without the hidden bit):
1 10000111 00101100
6. The mantissa bits are more than 4 bits (mantissa available bits); rounding is needed. If we applied the truncation rounding mode then the stored value is:
1 10000111 0010.

In this paper, we present the design of a floating-point multiplier that does not include built-in rounding support. Instead, rounding can be implemented as a separate functional unit, which may be accessed by either the multiplier or a floating-point adder. This modular approach enables higher precision, particularly when the multiplier is directly coupled with an adder in a Multiply and Accumulate (MAC) unit. The structure of the multiplier is illustrated in Fig. 2. The operations exponent addition, significand multiplication, and sign computation—are performed independently and in parallel to optimize performance. The significand multiplication is carried out on two 24-bit operands (including the implicit leading 1), producing a 48-bit intermediate product (IP). This IP is represented as bits [47:0], with the

ISSN: 2582 - 6379
IJISEA Publications
International Journal for Interdisciplinary Sciences and Engineering Applications
IJISEA - An International Peer- Reviewed, Open Access Journal
2025, Volume 6 Issue 3
www.ijisea.org

binary point positioned between bits 46 and 45. The subsequent sections describe each functional block of the floating-point multiplier in detail.

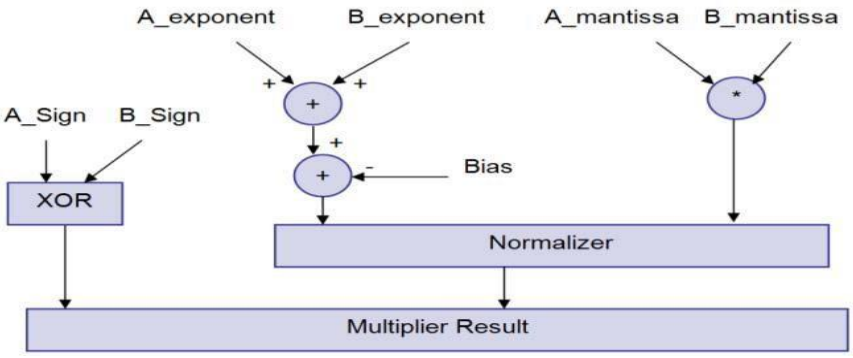


Figure 2: Floating point multiplier block diagram

III. HARDWARE OF FLOATING-POINT MULTIPLIER

A. Sign bit calculation

Multiplying two numbers results in a negative sign number iff one of the multiplied numbers is of a negative value. By the aid of a truth table, we find that this can be obtained by XORing the sign of two inputs.

B. Unsigned Adder (for exponent addition)

The exponent adder is responsible for calculating the sum of the exponents of the two input operands and subtracting the bias value (127), as per the IEEE 754 standard. This operation can be expressed as:

$$\text{Intermediate Exponent} = A_{\text{exponent}} + B_{\text{exponent}} - \text{Bias}$$

The addition is performed on 8-bit values, and since the significand multiplication (24-bit × 24-bit) dominates the overall computation time, a high-speed adder for the exponent is not critical. Therefore, a moderate-speed 8-bit ripple carry adder is used for this purpose. As illustrated in Fig. 3, a ripple carry adder is composed of a chain of full adders (and one-half adder for the least significant bit). Each full adder takes three inputs—two operand bits (A, B) and a carry-in (Ci)—and produces a sum (S) and a carry-out (Co). The carry-out from each stage is propagated ("rippled") to the next stage in the chain, completing the 8-bit addition.

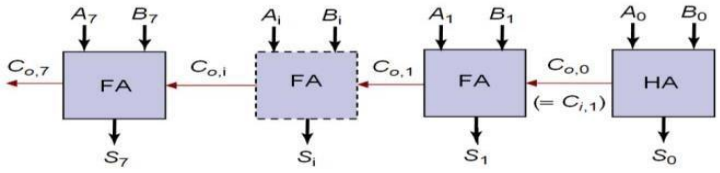


Figure 3: Ripple Carry Adder

ISSN: 2582 - 6379
IJISEA Publications
International Journal for Interdisciplinary Sciences and Engineering Applications
IJISEA - An International Peer- Reviewed, Open Access Journal
2025, Volume 6 Issue 3
www.ijisea.org

The addition process produces an 8-bit sum (S7 to S0) and acarry bit (Co,7). These bits are concatenated to form a 9-bit addition result (S8 to S0) from which the Bias is subtracted. The Bias is subtracted using an array of ripple borrow subtractors.

A normal subtractor has three inputs (minuend (S), subtrahend (T), Borrow in (Bi)) and two outputs (Difference (R), Borrow out (Bo)). The subtractor logic can be optimized if one of its inputs is a constant value which is our case, where the Bias is constant (127|10 = 001111111|2).

Table 1 shows the truth table for a 1-bit subtractor with the input T equal to 1 which we will call "one subtractor (OS)"

Table 1: 1-Bit Subtractor With The Inut T=1

S	T	B_i	Difference(R)	B_o
0	1	0	1	1
1	1	0	0	0
0	1	1	0	1
1	1	1	1	1

The Boolean Equation (2) and (3) represent this subtractor:

$$Difference(R) = \overline{S \oplus B_i} \quad (2)$$

$$Borrow\text{out (Bo)} = \overline{S} + B_i \quad (3)$$

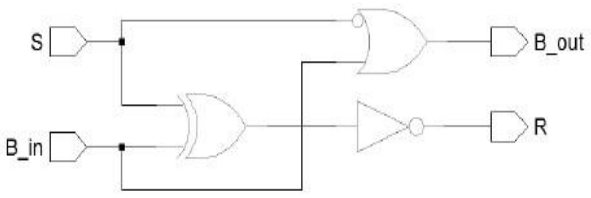


Figure 4: 1-bit subtractor with the input T = 1

Table 2 shows the truth table for a 1-bit subtractor with the input T equal to 0 which we will call "zero subtractor (ZS)"

Table 2. 1-Bit Subtractor With The Inut T=0

S	T	B_i	Difference(R)	B_o
0	0	0	0	0
1	0	0	1	0
0	0	1	1	1
1	0	1	0	0

ISSN: 2582 - 6379
IJISEA Publications
International Journal for Interdisciplinary Sciences and Engineering Applications
IJISEA - An International Peer- Reviewed, Open Access Journal
2025, Volume 6 Issue 3
www.ijisea.org

The Boolean Equation (4) and (5) represent this subtractor:

$$Difference(R) = S \oplus B_i \quad (4)$$

$$Borrowout\ (Bo) = \overline{S} \cdot B_i \quad (5)$$

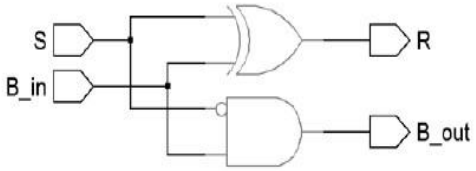


Figure 5: 1-bit subtractor with the input T = 0

Fig. 6 shows the Bias subtractor which is a chain of 7 one subtractors (OS) followed by 2 zero subtractors (ZS); the borrow output of each subtractor is fed to the next subtractor. If an underflow occurs then Eresult < 0 and the number is out of the IEEE 754 single precision normalized numbers range; in this case the output is signaled to 0 and an underflow flag is asserted.

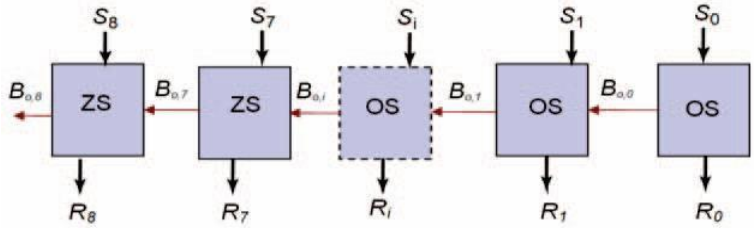


Figure 6: Ripple Borrow Subtractor

C. Unsigned Multiplier (for significand multiplication)

This unit is responsible for multiplying the unsigned significand and placing the decimal point in the multiplication product. The result of significand multiplication will be called the intermediate product (IP). The unsigned significand multiplication is done on 24 bits. Multiplier performance should be taken into consideration so as not to affect the whole multiplier's performance. A 24x24 bit carry save multiplier architecture is used as it has a moderate speed with a simple architecture. In the carry save multiplier, the carry bits are passed diagonally downwards (i.e. they carry bit is propagated to the next stage). Partial products are made by ANDing the inputs together and passing them to the appropriate adder.

Carry save multiplier has three main stages:

1- The first stage is an array of half adders.
2- The middle stages are arrays of full adders. The number of middle stages is equal to the significand size minus two.
3- The last stage is an array of ripple carries adders. This stage is called the vector merging stage.

ISSN: 2582 - 6379
IJISEA Publications
International Journal for Interdisciplinary Sciences and Engineering Applications
IJISEA - An International Peer- Reviewed, Open Access Journal
2025, Volume 6 Issue 3
www.ijisea.org

The number of adders (Half adders and Full adders) in each stage is equal to the significand size minus one. For example, a 4x4 carry save multiplier is shown in Fig. 7 and it has the following stages:
1- The first stage consists of three half adders.
2- Two middle stages; each consists of three full adders.
3- The vector merging stage consists of one-half adder and two full adders.
The decimal point is between bits 45 and 46 in the significand multiplier result. The multiplication time taken by the carry save multiplier is determined by its critical path. The critical path starts at the AND gate of the first partial products (i.e. a1b0 and a0b1), passes through the carry logic of the first half adder and the carry logic of the first full adder of the middle stages, then passes through all the vector merging adders. The critical path is marked in bold in Fig. 7

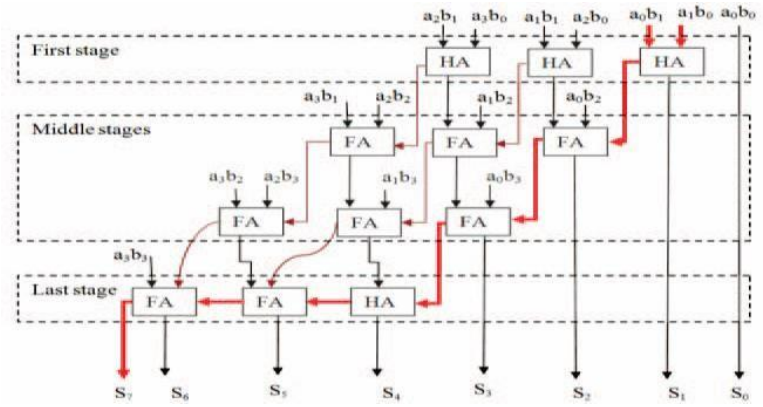


Figure 7: 4x4 bit Carry Save multiplier

In Fig. 7:
1- Partial product: aibj = ai and bj
2- HA: half adder
3- FA: full adder

D. Normalizer

The result of the significand multiplication (intermediate product) must be normalized to have a leading '1' just to the left of the decimal point (i.e. in the bit 46 in the intermediate product). Since the inputs are normalized numbers then the intermediate product has the leading one at bit 46 or 47 1- If the leading one is at bit 46 (i.e. to the left of the decimal point) then the intermediate product is already a normalized number and no shift is needed.

2- If the leading one is at bit 47 then the intermediate product is shifted to the right and the exponent is incremented by 1.

The shift operation is done using combinational shift logic made by multiplexers. Fig. 8 shows a simplified logic of a Normalizer that has an 8-bit intermediate product input and a 6-bit intermediate exponent input.

ISSN: 2582 - 6379
IJISEA Publications
International Journal for Interdisciplinary Sciences and Engineering Applications
IJISEA - An International Peer- Reviewed, Open Access Journal
2025, Volume 6 Issue 3
www.ijisea.org

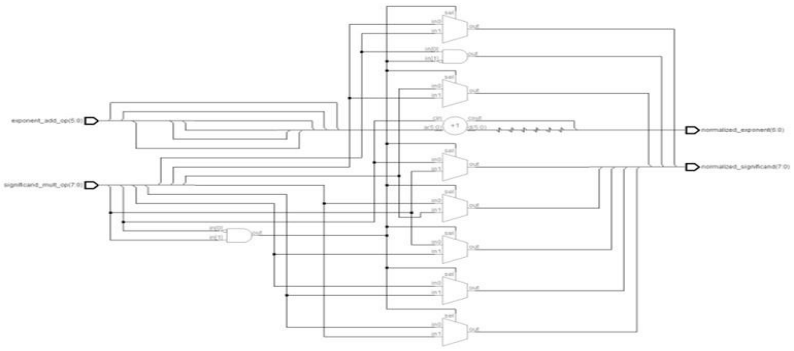


Figure 8: Simplified Normalizer logic

IV. UNDERFLOW/OVERFLOW DETECTION

Overflow/underflow means that the result's exponent is too large/small to be represented in the exponent field. The Exponent of the result must be 8 bits in size, and must be between 1 and 254 otherwise the value is not a normalized one. Between 1 and 254 otherwise the value is not a normalized one. An overflow may occur while adding the two exponents or during normalization. Overflow due to exponent addition may be compensated during subtraction of the bias; resulting in a normal output value (normal operation). An underflow may occur while subtracting the bias to form the intermediate exponent. If the intermediate exponent < 0 then it's an underflow that can never be compensated; if the intermediate exponent = 0 then it's an underflow that may be compensated during normalization by adding 1 to it.

$$Eresult = E1 + E2 - 127 \qquad (6)$$

E1 and E2 can have the values from 1 to 254; resulting in Eresult having values from -125 (2-127) to 381 (508-127); but for normalized numbers, Eresult can only have the values from 1 to 254. Table 3 summarizes the Eresult different values and the effect of normalization on it.

Table 3: Normalization Effect on Result's Exponent And Overflow/Underflow Detection

E_{result}	Category	Comments
$-125 \leq E_{result} < 0$	Underflow	Can't be compensated during normalization
$E_{result} = 0$	Zero	May turn to normalized number during normalization (by adding 1 to it)
$1 < E_{result} < 254$	Normalized number	May result in overflow during normalization
$255 \leq E_{result}$	Overflow	Can't be compensated

ISSN: 2582 - 6379
IJISEA Publications
International Journal for Interdisciplinary Sciences and Engineering Applications
IJISEA - An International Peer- Reviewed, Open Access Journal
2025, Volume 6 Issue 3
www.ijisea.org

5. PIPELINING THE MULTIPLIER

In order to enhance the performance of the multiplier, three pipelining stages are used to divide the critical path thus increasing the maximum operating frequency of the multiplier. The pipelining stages are imbedded at the following locations:

1. In the middle of the significand multiplier, and in the middle of the exponent adder (before the bias subtraction).
2. after the significand multiplier, and after the exponent adder.
3. At the floating-point multiplier outputs (sign, exponent and mantissa bits).

Fig. 9 shows the pipelining stages as dotted lines.

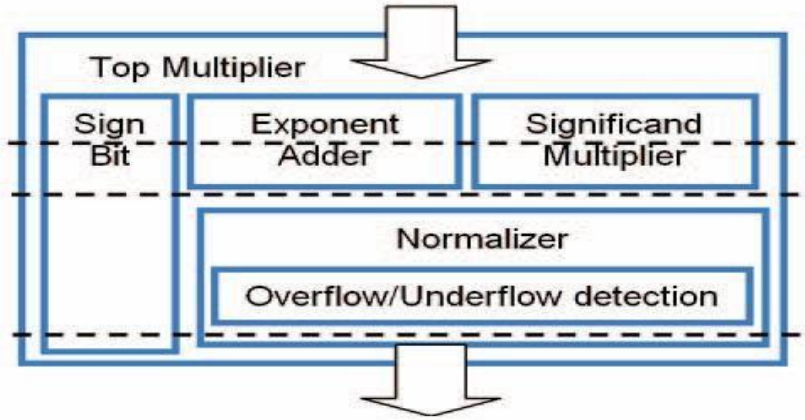


Figure 9: Floating point multiplier with pipelined stages

Three pipelining stages mean that there is latency in the output by three clocks. The synthesis tool "retiming" option was used so that the synthesizer uses its optimization logic to better place the pipelining registers across the critical path.

V. RESULT

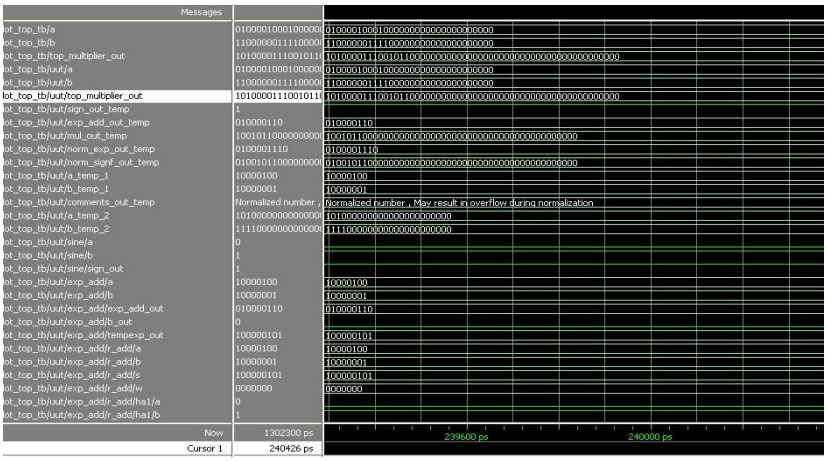


Figure 10: Simulation result of top multiplier

ISSN: 2582 - 6379
IJISEA Publications
International Journal for Interdisciplinary Sciences and Engineering Applications
IJISEA - An International Peer- Reviewed, Open Access Journal
2025, Volume 6 Issue 3
www.ijisea.org

Fig.11 shows the RTL diagram of top multiplier which is having inputs A, B of 32 bit and output of 56bit.



Figure 11: RTL diagram of top multiplier

VI. IMPLEMENTATION AND TESTING

The whole multiplier (top unit) was tested against the Xilinx floating point multiplier core generated by Xilinx coregen. Xilinx core was customized to have two flags to indicate overflow and underflow, and to have a maximum latency of three cycles. Xilinx core implements the "round to nearest" rounding mode.

The area of Xilinx core is less than the implemented floating-point multiplier because the latter doesn't truncate/round the 48 bits result of the mantissa multiplier which is reflected in the amount of function generators and registers used to perform operations on the extra bits; also, the speed of Xilinx core is affected by the fact that it implements the round to nearest rounding mode.

VII. CONCLUSIONS AND FUTURE WORK

This paper presents the implementation of a floating-point multiplier compliant with the IEEE 754 binary interchange format. The design omits rounding and directly outputs the full 48-bit result of the significand multiplication. This approach enhances precision when the output is subsequently used in downstream units such as a floating-point adder in a Multiply and Accumulate (MAC) architecture. The multiplier is designed with a three-stage pipelined architecture and is implemented on a Xilinx Virtex-5 FPGA. Post-synthesis, the design achieves a performance of 301 MFLOPs.

REFERENCES:

[1] IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic, 2018.
[2] B. Fagin and C. Renard, "Field Programmable Gate Arrays and Floating-Point Arithmetic," IEEE Transactions on VLSI, vol. 2, no. 3, pp. 365–367, 2020.
[3] Naga Jyothi, Grande, Kundu Debanjan, and Gorantla Anusha. "ASIC implementation of fixed-point iterative, parallel, and pipeline CORDIC algorithm." Soft Computing for Problem Solving: SocProS 2018, Volume 1. Singapore: Springer Singapore, 2019. 341-351.

ISSN: 2582 - 6379
IJISEA Publications
International Journal for Interdisciplinary Sciences and Engineering Applications
IJISEA - An International Peer- Reviewed, Open Access Journal
2025, Volume 6 Issue 3
www.ijisea.org

[4] L. Louca, T. A. Cook, and W. H. Johnson, “Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs,” Proceedings of 83 the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM’96), pp. 107–116, 2022.

[5] A. Jaenicke and W. Luk, "Parameterized Floating-Point Arithmetic on FPGAs", Proc. of IEEE ICASSP, 2001, vol. 2, pp.897-900.

[6] B. Lee and N. Burgess, “Parameterisable Floating-point Operations on FPGA,” Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems, and Computers, 2022

[7] “DesignChecker User Guide”, HDL Designer Series 2010.2a, Mentor Graphics, 2020

[8] “PrecisionR Synthesis User’s Manual”, Precision RTL plus 2020a update 2, Mentor Graphics, 2020.

[9] Patterson, D. & Hennessy, J. (2025), Computer Organization and Design: The Hardware/software Interface, Morgan Kaufmann.

[10] NagaJyothi, Grande, and Sriadibhatla SriDevi. "Distributed arithmetic architectures for fir filtersa comparative review." 2017 International conference on wireless communications, signal processing and networking (WiSPNET). IEEE, 2017.

[11] Sharma, Abhay, and Tarun Kumar Rawat. "Truncated Wallace Based Single Precision Floating Point Multiplier." 2018 7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO). IEEE, 2018.

[12] Ramya, V., and R. Seshasayanan. "Low power single precision BCD floating–point Vedic multiplier." Microprocessors and Microsystems 72 (2020): 102930.

[13] Avanija, J., et al. "Designing a fuzzy Q-learning power energy system using reinforcement learning." International Journal of Fuzzy System Applications (IJFSA) 11.3 (2022): 1-12.

[14] DiCecco, Roberto, Lin Sun, and Paul Chow. "FPGA-based training of convolutional neural networks with a reduced precision floating-point library." 2017 International Conference on Field Programmable Technology (ICFPT). IEEE, 2017.

[15] Anuhya, Pasupuleti, and R. Dhanabal. "Asic Implementation of Efficient Floating Point Multiplier." 2018 4th International Conference on Electrical Energy Systems (ICEES). IEEE, 2018.