Accuracy and Hardware Cost Analysis of Multi-Format Floating-Point Arithmetic Generated by FloPoCo on FPGA

Master's thesis in Embedded Electronic System Design

Boyue Sun & Zhili Xia

Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026

Master's thesis 2026

Accuracy and Hardware Cost Analysis of Multi-Format Floating-Point Arithmetic Generated by FloPoCo on FPGA

Boyue Sun & Zhili Xia

Department of Microtechnology and Nanoscience
Chalmers University of Technology
Gothenburg, Sweden 2026

Accuracy and Hardware Cost Analysis of Multi-Format Floating-Point Arithmetic Generated by FloPoCo on FPGA
Boyue Sun & Zhili Xia

Supervisor: Per Larsson-Edefors, Department of Microtechnology and Nanoscience
Examiner: Lena Peterson, Department of Microtechnology and Nanoscience

Master's Thesis 2026
Department of Microtechnology and Nanoscience
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2026

Accuracy and Hardware Cost Analysis of Multi-Format Floating-Point Arithmetic Generated by FloPoCo on FPGA
Boyue Sun & Zhili Xia
Department of Microtechnology and Nanoscience
Chalmers University of Technology

Abstract

Modern artificial intelligence (AI) and digital signal processing (DSP) workloads are highly data-driven and computationally demanding, relying heavily on massive multiply-accumulate (MAC) operations. While standard IEEE 754 floating-point arithmetic provides a vast dynamic range, its strict compliance requirements, such as subnormal handling and exact rounding, incur significant hardware overhead.

To address this, this thesis evaluates the accuracy and hardware cost of multi-format floating-point arithmetic generated by the FloPoCo framework on field-programmable gate arrays (FPGAs). We employ a hardware-software co-simulation methodology, combining Xilinx Vivado for power, performance, and area assessment with a Python-based error evaluation engine using Gaussian distributed test vectors to emulate AI workloads.

Our results demonstrate that FloPoCo's custom Nfloat format, which eliminates subnormal support and utilizes a dedicated exception field, significantly reduces pipeline depth, look-up table (LUT) consumption, and dynamic power compared to IEEE 754 implementations across all tested bit-widths. Furthermore, a comparative analysis between a unified-precision Nfloat MAC and an IEEE fused multiply-add (FMA) reveals that the NFloat MAC achieves over 50% power and area savings while maintaining identical algorithmic fidelity at low-to-medium precisions. Finally, we investigate the performance of mixed-precision MAC architectures in deep accumulation chains with lengths up to 5120 accumulation steps. The results show that mixed-precision computation effectively maintains a stable relative error near 0.001%. FloPoCo and the Nfloat format present a efficient and customizable alternative for FPGA-based high-performance computing.

Keywords: FPGA, Floating-Point Arithmetic, FloPoCo, Nfloat, Multiply-Accumulate, Mixed-Precision, Hardware-Software Co-Simulation

Acknowledgements

We would like to express our deepest gratitude to our supervisor, Per Larsson-Edefors, for his invaluable guidance, patience, and continuous support throughout the duration of this Master's thesis at Chalmers University of Technology. His profound expertise and insightful feedback have been instrumental in shaping the direction and quality of this research. We are also very grateful to our examiner, Lena Peterson, for her writing guidance and for providing the academic advice that made this project possible.

We would also like to thank doctoral student Xu Wang for his assistance during the project. Thank you everyone for your support. It not only kept us motivated but also allowed us to deeply experience Swedish culture, making our time in Sweden unforgettable.

Boyue Sun & Zhili Xia, Gothenburg, June 2026

Contents

List of Abbreviations

AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
BRAM	Block RAM
CLB	Configurable Logic Block
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DSP	Digital Signal Processing
FF	Flip-Flop
FMA	Fused Multiply-Add
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GEMM	General Matrix Multiplication
HDL	Hardware Description Language
LUT	Look-Up Table
MAC	Multiply-Accumulate
MRE	Mean Relative Error
NaN	Not a Number
PPA	Power, Performance, and Area
RAM	Random Access Memory
ROM	Read-Only Memory
RTL	Register-Transfer Level
SAIF	Switching Activity Interchange Format
SRL	Shift Register Lookup
THS	Total Hold Slack
TNS	Total Negative Slack
ULP	Unit in the Last Place
VCD	Value Change Dump
WHS	Worst Hold Slack
WNS	Worst Negative Slack

1

Introduction

Artificial intelligence (AI) and digital signal processing (DSP) form the underlying backbone of modern technologies. From smart devices and high-speed communication networks to autonomous driving and advanced healthcare diagnostics, these technologies are reshaping our daily lives. However, with the increasing demand for AI training and DSP data processing, it is clear that traditional general-purpose processor hardware has become increasingly disadvantaged in this process, making it difficult to cope with the growing data processing needs. This situation has led researchers to seek solutions to this computing need by utilizing the high parallelism of heterogeneous computing platforms as complement to central processing units (CPUs).

Both AI and DSP are data-driven and computationally demanding fields. Executing complex digital filters or training deep neural networks involves billions of mathematical operations [1]. The efficacy of these algorithms relies heavily on how the underlying hardware represents and processes numerical data. In these domains, numerical values exhibit extreme variations in magnitude—ranging from massive feature accumulations to the minuscule gradients generated during backpropagation or the extraction of faint signals [2]. Traditional integer or fixed-point formats, with their rigid representational windows, are inherently susceptible to overflow or precision loss when handling such data [3].

Unlike integer or fixed-point representations, floating-point formats offer a vast dynamic range. This flexibility is important for modern algorithms. In AI, a wide dynamic range prevents gradient underflow during backpropagation and smoothly accommodates the highly variable distributions of neural network weights and activations. In DSP applications, floating-point arithmetic ensures high signal-to-noise ratios and effectively prevents numerical overflow during operations like recursive filtering or fast Fourier transforms.

In recent years, with the rapid development of AI and DSP applications, the scale of data that needs to be processed has begun to increase rapidly. At the same time, larger-scale models and more complex algorithms have significantly increased the computational demands required for machine learning. Emerging applications such as speech recognition and image recognition, place high demands on real-time computation. While general-purpose processors such as CPUs and GPUs provide flexibility and programmability, they often suffer from limited energy efficiency and suboptimal performance when handling highly parallel numerical workloads. In contrast, hardware accelerators based on field-programmable gate arrays (FPGA)

and application-specific integrated circuits (ASIC) offer significant advantages due to their ability to exploit fine-grained data-level parallelism, deep pipelining, and application-specific optimization. FPGA is a semi-custom circuit chip that overcomes the shortcoming of custom specific circuits, which cannot be modified once they are fabricated. It enables designers to tailor arithmetic units and datapaths to specific computational patterns, making them well-suited for floating-point intensive tasks such as matrix multiplication, convolution, and streaming signal processing. ASICs further push this specialization by achieving superior performance-per-watt through fully customized architectures.

Compared with general-purpose processors, FPGA and ASIC architectures achieve higher performance-per-watt due to specialization [4]. As a result, modern computing systems increasingly rely on FPGA- and ASIC-based accelerators to meet the computational demands of large-scale AI models and real-time DSP [5]. This trend highlights the critical role of specialized hardware platforms in advancing efficient floating-point computation and underscores their growing importance in both academia and industry. To better meet the demands for computing performance, many large companies worldwide have invested in and implemented FPGA acceleration hardware. Intel acquired Altera in 2015, and AMD completed its acquisition of Xilinx in 2022. Other countries and companies have also begun developing their own FPGA chips and architectures [6],[7],[8]. All of this demonstrates the broad application prospects and market expectations for FPGAs in the current environment.

The design and implementation of digital systems on FPGA and ASIC platforms typically rely on hardware description languages (HDLs) such as Verilog and VHDL. The development process involves multiple stages, including register-transfer level (RTL) design, functional simulation, logic synthesis, placement and routing, and timing verification. Each stage requires careful consideration of both functional correctness and hardware constraints, making the overall workflow significantly more complex than software development.

In particular, designing arithmetic units such as floating-point operators at the RTL level is a non-trivial task. These operators require the implementation of sophisticated mechanisms including operand alignment, normalization, rounding, and exception handling, all of which must be carefully optimized for performance and resource efficiency. Furthermore, achieving timing closure—ensuring that the design meets a given clock frequency constraint—often involves iterative refinement of the architecture, pipeline structure, and datapaths. This process can be time-consuming and demands substantial expertise in digital design and hardware optimization.

As a result, the development of efficient floating-point hardware on FPGA or ASIC platforms typically requires significant engineering effort and long design cycles, especially when exploring multiple design configurations or precision formats. This need has made a tool that can generate highly customized, synthesizable HDL hardware code for floating-point operators based on the floating-point format, target hardware type, and frequency requirements an urgent necessity. Such a tool could greatly accelerate development and reduce development costs for developers.

To address the growing demand for customized arithmetic in FPGA and ASIC designs, researchers developed FloPoCo (Floating-Point Cores) [9],[10]. FloPoCo is

an open-source generator that produces synthesizable VHDL code for a wide range of arithmetic operators. By allowing developers to freely specify functional parameters (such as exponent and mantissa bitwidths) and performance constraints (such as target frequency and pipeline depth), it can significantly accelerate the hardware design cycle.

Besides automated code generation, FloPoCo integrates a robust verification framework. Since customized operators encompass an infinite number of parameter combinations, relying on standard, pre-computed verification methods is insufficient. FloPoCo solves this by automatically generating a testbench and corresponding test vectors alongside the VHDL code. It derives these bit-accurate test vectors directly from strict mathematical specifications using reference multiple-precision libraries like MPFR [11]. This approach provides a baseline testing option for standalone operators to evaluate the exact mathematical intent rather than just mirroring the generated hardware logic, granting the accuracy of the verification.

A key architectural advantage of FloPoCo is its flexible support for numerical data types. It fully implements a parameterized version of the IEEE-754 floating-point standard, allowing arbitrary exponent and mantissa sizes while using traditional exponent encoding to handle infinities, not a number (NaN), and subnormal numbers. However, recognizing the severe hardware overhead associated with these edge cases, FloPoCo introduces an optimized, hardware-friendly alternative known as the Nfloat (or FloPoCo) format [12]. Unlike the IEEE standard, Nfloat appends a dedicated two-bit exception field to explicitly flag zeroes, infinities, and NaNs. This design choice frees up the entire exponent range for normal numbers and eliminates the complex control logic required to decode exceptional cases from the exponent. Furthermore, Nfloat deliberately drops support for subnormal numbers, prioritizing computational density and hardware efficiency over edge-case precision. This makes the Nfloat format highly attractive for performance-critical custom accelerators.

1.1 Related Work

Due to the high overhead of IEEE 754, researchers have developed formats with various precision levels.

High cost of IEEE 754

Although the IEEE 754 single-precision (FP32) format is widely adopted [13], it also comes with a heavy hardware cost. To guarantee high numerical accuracy, the FP32 standard allocates 23 bits to the mantissa (fraction). However, in digital circuit design, the physical area of a hardware multiplier scales quadratically with the mantissa width [14]. Consequently, processing a 23-bit mantissa requires massive silicon real estate.

Furthermore, the IEEE 754 standard supports for subnormal (or denormal) numbers to ensure gradual underflow when values approach zero. Processing these edge-case values forces hardware designers to implement complex exception-handling logic, additional shifters, and normalization circuits [15]. This architectural complexity

increases power consumption and datapath latency. In computationally intensive fields like AI and DSP, which have a high tolerance for reduced precision, this hardware overhead becomes redundant. It limits the number of multiply-accumulate (MAC) units and arithmetic logic units (ALUs) that engineers can pack onto a single processor, ultimately restricting the peak computational throughput of the hardware [16].

Methods using low-precision

To deal with the growing computational demand and high hardware cost that traditional high-precision floating-point arithmetic brings, reduced-precision formats floating-point have been widely used in modern AI systems. Unlike the traditional IEEE 754 formats, low-precision formats trade off some degree of precision by cutting the length of fraction bits in exchange for significant improvements in hardware efficiency and computational throughput. This trend is motivated by the observation that many AI workloads, especially deep neural networks, have a high tolerance to numerical errors but still require a large dynamic range, so full IEEE-754 precision is not required to achieve good performance.

As a result, several industry-driven formats have been proposed and widely deployed by companies in the industry. For instance, Bfloat16 [17], introduced by Google, retains the same exponent width as single-precision floating point while significantly reducing mantissa bits, enabling efficient training and inference. Similarly, TensorFloat-32 which was developed by NVIDIA, reduces fraction precision to accelerate matrix operations on GPUs without requiring major changes to existing software [18]. Recently, even lower-precision formats such as FP8 have been explored to further improve performance and energy efficiency [19]. These developments demonstrate a trend toward precision scalability in floating-point computation where different levels of precision can be selected based on application requirements. This shift motivates the exploration of flexible and customizable floating-point formats, which can potentially achieve a better balance between numerical accuracy and hardware cost compared to fixed IEEE standards.

1.2 Purpose and Goal

This study focuses on the VHDL code generated by FloPoCo, with the core research questioning whether the Nfloat format provided by FloPoCo can achieve the computational correctness and accuracy achievable by IEEE 754 while maintaining structural advantages.

The primary focus is on the performance of generated RTL codes, particularly comparing the performance of Nfloat and the same type of operators in the IEEE 754 format. The research aims to determine whether the operator code generated by FloPoCo can correctly perform floating-point calculations with various exponent and mantissa lengths, and what the calculation errors are under different mantissa conditions. Most importantly, it observes whether its proprietary floating-point format, Nfloat, can achieve the same precision as the IEEE 754 format, and whether

Nfloat can simplify circuit logic and reduce resource usage.

Next, we will investigate how the MAC operation in the Nfloat format, constructed from code generated by FloPoCo, can achieve the computational accuracy of the fused multiply-add (FMA) in the IEEE format in both single operation and multiple accumulation processes, while simultaneously optimizing logic resources. In this process, we will also introduce a comparison between single-format and mixed-precision MACs to explore whether the code generated using FloPoCo can maintain the advantages of computational accuracy and simplified hardware resources under this computational structure.

1.3 Thesis Outline

This thesis is organized as follows.

Chapter 2 introduces the technical background of this work, including floating-point formats, floating-point arithmetic, MAC operations, FPGA resources, and numerical error metrics.

Chapter 3 presents the methodology used in this project. It describes the co-simulation flow, operator generation, Vivado-based synthesis and implementation, test stimulus generation, and numerical error evaluation.

Chapter 4 describes the design of the evaluated floating-point operators and the experimental framework. It introduces the FloPoCo-generated IEEE 754 and Nfloat operators, the manually constructed Nfloat MAC, and the testbench structure used for accumulation-chain evaluation.

Chapter 5 presents the experimental results. It compares different operators and precision configurations in terms of pipeline depth, resource utilization, power consumption, timing feasibility, and computational error.

Chapter 6 discusses the main findings and limitations of the project. It summarizes the advantages and limitations of FloPoCo and Nfloat, and reflects on the restrictions of the current evaluation setup.

Chapter 7 concludes the thesis by summarizing the main contributions and results, and outlines possible directions for future work.

2

Technical Background

This chapter provides an overview of the key concepts and technologies relevant to this thesis, including floating-point arithmetic, multiply-accumulate operations, FPGA hardware resources, and error evaluation metrics. This overview establishes the theoretical foundation necessary to design custom arithmetic architectures and evaluate their numerical precision.

2.1 Floating-Point

This section introduces the format, precision, calculation methods, and rounding methods of floating-point numbers.

2.1.1 Floating-Point Formats

Floating-point representation is the cornerstone of numerical computing, offering a wider dynamic range than fixed-point arithmetic. The IEEE 754 standard dominates this space, defining formats like single precision (IEEE32) shown in Figure 2.1(a) and double precision (IEEE64) [13]. However, for custom hardware accelerators, researchers often rely on alternative formats tailored for digital circuits, such as FloPoCo's Nfloat [9]. While IEEE 754 encodes exceptional values (like zeroes, infinities, and NaNs) using reserved exponent values, Nfloat explicitly allocates a dedicated two-bit exception field [12], such as Nfloat34 shown in Figure 2.1(b), Nfloat18 shown in Figure 2.1(c). This hardware-centric design eliminates complex exponent decoding logic and frees up the entire exponent range for normal numbers. Furthermore, to address the specific computational demands of modern artificial intelligence workloads, researchers frequently employ brain floating point (Bfloat16) [17], as shown in Figure 2.1(d). Originally developed by Google Brain for deep learning accelerators, Bfloat16 deliberately preserves the 8-bit exponent of standard IEEE 754 single precision to maintain a massive dynamic range. Meanwhile, it truncates the mantissa to 7 bits, representing a optimized trade-off that significantly reduces hardware complexity.

A standard floating-point number consists of three components: a sign bit (s), an exponent (E), and a mantissa or fraction (F). The hardware evaluates the numerical value using the formula

$$X = (-1)^s \times 2^{E-E_0} \times (1.F) \tag{2.1}$$

where E_0 represents the exponent bias.

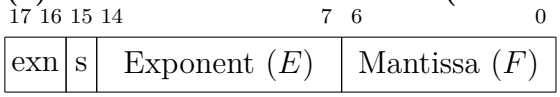
Beyond normal numbers, the standard defines "non-numbers" such as (NaN) for undefined operations (e.g., 0/0) and Infinity for computation overflows. It also specifies subnormal numbers (or denormals), which handle values extremely close to zero. Subnormals use a modified formula to provide gradual underflow [13].

$$X = (-1)^s \times 2^{E_{min}} \times (0.F) \tag{2.2}$$

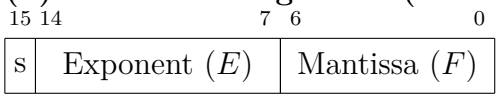
(a) IEEE 754 Single Precision (IEEE32)

(b) FloPoCo Nfloat Format (Nfloat34)

(c) FloPoCo Nfloat Format (Nfloat18)



(d) Brain Floating Point (Bfloat16)



(e) Posit Format (Posit32)

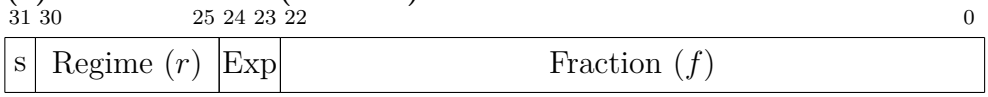


Figure 2.1: Data layouts of standard IEEE 754 formats, custom FloPoCo Nfloat formats, and brain floating point. The Nfloat format introduces a 2-bit exception field (exn) to explicitly encode zeroes, infinities, and NaNs.

However, processing subnormals forces digital circuits to implement costly exception-handling datapaths, increasing silicon area and power consumption [15],[20]. Consequently, custom formats like Nfloat explicitly drop subnormal support to maximize hardware efficiency.

As an alternative to conventional floating-point standards, the Posit number system introduces a dynamic tapered-precision mechanism [21] shown in Figure 2.1(e). Unlike standard IEEE 754 or custom Nfloat formats that rely on fixed-width exponent and mantissa fields, a Posit format dynamically partitions its bitwidth using a run-length encoded *regime* field. This unique structural design grants Posits superior numerical accuracy near ± 1.0 and a wider dynamic range, effectively overcoming the limitations of subnormal numbers and overflow handling in traditional floating-point arithmetic. However, from a hardware perspective, decoding the variable-length

regime field requires complex leading-zero detectors and dynamic shifters. This increases the critical path delay and area overhead within the arithmetic datapaths, which poses challenges for high-frequency pipelined FPGA implementations.

2.1.2 Floating-Point Numbers

The bitwidth allocation between the exponent and mantissa fields fundamentally dictates a format's numerical behavior. The exponent determines the dynamic range of the format. To represent both microscopic fractions and massive magnitudes, the exponent uses a biased representation by subtracting a constant bias (E_0) from the stored unsigned integer. A difference between the formats lies in their utilization of the exponent space. Standard IEEE 754 sacrifices the all-zero ($00\ldots00$) and all-one ($11\ldots11$) exponent bit patterns to encode subnormals, zeroes, infinities, and NaNs [13]. Nfloat, by its dedicated exception bits, reclaims these boundary patterns for normal computation [12]. This architectural shift expands the valid dynamic range for normal numbers without requiring extra exponent bits.

The Mantissa field determines the precision, representing the resolution between two consecutive floating-point numbers. In digital circuit design, truncating the mantissa reduces the physical size of hardware multipliers quadratically [22][23], which serves as the core motivation behind modern low-precision formats like Bfloat16.

2.1.3 Hardware Computation Methods

Implementing floating-point arithmetic in digital circuits requires coordinating complex mathematical operations in datapaths. For addition and subtraction, when two operands share the same floating-point format (e.g., both possessing an 8-bit exponent field), their actual exponent values typically differ. Fundamentally, floating-point circuits execute fixed-point operations on the mantissas internally. However, the strict prerequisite for this internal fixed-point addition is that both mantissas must operate at the exact same order of magnitude. To resolve this discrepancy, the floating-point hardware executes an alignment phase that calculates the absolute difference between the two exponent values, and then right-shifts the mantissa of the smaller number by this exact difference. This rightward shift proportionally scales down the mantissa while implicitly incrementing its exponent until it perfectly matches the larger exponent. Only after this exact alignment is achieved does the floating-point ALU perform addition or subtraction on the mantissas. Finally, the circuit normalizes the result by shifting the output mantissa and adjusting the newly computed exponent to conform strictly to the standard format [24].

Multiplication is logically simpler but highly consuming hardware resources. The circuit adds the two exponents (subtracting the bias) and simultaneously multiplies the two mantissas using a large integer multiplier. Finally, it normalizes the resulting product [25].

2.1.4 Rounding Methods

During arithmetic operations, the hardware inevitably generates intermediate results that exceed the target mantissa bitwidth. The circuit must round these values to fit the destination format.

The IEEE 754 standard strictly mandates Correct Rounding across all its specified rounding modes (e.g., Round to Nearest, Round toward Zero, and Round toward Infinity) [13]. The concept of correct rounding requires the hardware to return a result exactly as if the computation were performed with infinite precision before applying the respective rounding rule. To guarantee this mathematical rigor across any mode, the digital circuit must evaluate the exact truncated remainder using three auxiliary bits: the Guard bit, the Round bit, and the Sticky bit. The Sticky bit is a logical OR of all remaining discarded bits which requires a lot of hardware resources. Furthermore, executing the final rounding decision often triggers a carry-propagate addition across the entire mantissa. This terminal carry propagation severely extends the critical path, limiting the maximum achievable clock frequency [26, Chap. 9].

To address this architectural bottleneck, custom hardware accelerators and core generators like FloPoCo frequently adopt Faithful Rounding [12]. A faithfully rounded operator completely relaxes the strict IEEE mandate of exact boundary evaluation. It simply guarantees that the output is one of the two representable floating-point numbers immediately bounding the exact mathematical result. By accepting an error bound of one unit in the last place (ULP) rather than 0.5 ULP, faithful rounding eliminates the need for computing the sticky bit and performing the final carry-propagate addition [26, Sec. 8.2]. For domain-specific applications like deep learning or DSP—which are resilient to minor precision variations, this relaxation yields arithmetic datapaths that are cheaper, faster and smaller than their strict IEEE counterparts.

In this work, rounding is particularly important because the evaluated operators do not only differ in exponent and mantissa widths, but also in how intermediate results are reduced back to the target format. For floating-point addition, the alignment stage may shift the smaller mantissa to the right, producing discarded low-order bits. For multiplication, the product of two significands has approximately twice the original significand width. These extra bits cannot be directly stored in the destination format and therefore must be used to make a rounding decision before the final result is packed.

A commonly used rounding mode is round-to-nearest-even [13]. In this mode, the result is rounded to the nearest representable value. If the exact value lies exactly halfway between two representable values, the result whose least significant mantissa bit is even is selected. In hardware, this decision is usually implemented by checking the round bit, the sticky information of the remaining discarded bits, and the least significant bit of the retained mantissa. The result is incremented when the discarded part is greater than half an ULP, or when it is exactly half an ULP and the retained mantissa is odd. This tie-breaking rule avoids introducing a systematic upward or downward bias over many operations.

In the context of this work, rounding also explains the numerical difference between a fused multiply-add operator and a conventional multiply-accumulate structure. An IEEE FMA computes the product and the addition in an extended internal datapath and performs only one final rounding step. In contrast, the Nfloat MAC constructed from separate multiplier and adder operators first rounds the multiplication result to the target Nfloat format and then rounds again after the following addition. Therefore, even when the input and output formats have the same exponent and mantissa widths, the number and position of rounding steps may affect the accumulated numerical error.

2.2 Matrix Multiplication

Modern deep neural networks fundamentally rely on massive linear algebra computations. Whether extracting spatial features in convolutional neural networks (CNNs) or calculating attention scores in Transformer architectures, general matrix multiplication (GEMM) consistently dominates the workload. Hardware profiling of contemporary AI models reveals that matrix multiplications routinely consume over 90% of the total execution time during both training and inference [27]. Consequently, accelerating GEMM operations remains the primary design objective for AI hardware accelerator.

From a mathematical perspective, multiplying two matrices decomposes into a vast collection of vector dot products. At the hardware execution level, this translates into billions of MAC operations. A standard MAC operation computes the product of two inputs and adds the result to an accumulator register, fundamentally executing the equation $D = A \times B + C$. Because of this direct mapping, engineers typically measure an AI chip's peak computational throughput by the sheer number of physical MAC units it can integrate and sustain on a single silicon die.

To execute these operations with maximum efficiency, modern arithmetic datapaths frequently implement FMA units. While a traditional MAC operation might compute the multiplication, round the intermediate product, and then perform the addition, an FMA unit calculates the entire expression $A \times B + C$ as a single, indivisible operation. It performs only one final rounding step at the end. This hardware fusion not only reduces the datapath latency and silicon area but also eliminates the precision loss caused by intermediate rounding [26, Sec. 9.5].

2.3 Exponent and Fraction in Different Operators

From the preceding content, we can understand that the exponent bitwidth of floating-points mainly determines the numerical range of a floating-point number while the width of fraction bits determines the precision. However, based on our experience with addition and multiplication of fixed-point numbers, we can conclude that the results obtained from adding and multiplying two numbers separately can be quite different, and therefore the requirements for range and precision will also be very different [24].

In floating-point calculation, the multiplication can essentially be viewed as two parts, which are adding the exponent and multiplying the significant numbers. For two floating-point numbers x and y, which can be expressed as

$$x = (-1)^{S_x}(1.f_x)_2 2^{E_x} \tag{2.3}$$

$$y = (-1)^{S_y}(1.f_y)_2 2^{E_y} \tag{2.4}$$

the result can be expressed as

$$xy = (-1)^{S_x \oplus S_y} \cdot [(1.f_x)_2 (1.f_y)_2] \cdot 2^{E_x + E_y} \tag{2.5}$$

In this case, the exponent of new floating-point is

$$E_{out} = E_x + E_y + \text{normalization adjustment} \tag{2.6}$$

and we can conclude that if both numbers already have large exponents, multiplying them will result an even larger exponent which easily can exceed the representable limit:

$$E_{\text{out}} > E_{\max}$$

which we call overflow. Similarly, multiplying two very small numbers may result in the exponent of the product being less than the lower bound of representation, leading to underflow. Therefore, multiplication is very sensitive to the range of the exponent, especially in chain multiplication, dot product, and matrix multiplication, where intermediate results may be very large or very small.

As for fraction, the situation is different. Assuming the number of significant bits of both floating point is

$$p = wF + 1$$

the product of two numbers will theoretically produce a product close to $2p$ digits. However, the final result only retains about p bits allowed by the target format, so rounding is necessary. In this process, the multiplier first generates a longer fraction, which is then truncated and rounded to the target fraction length. This is the main source of error.

In a floating-point adder, the fractions are not added directly, but the exponents are needed to be aligned first. For the number x and y we mentioned before, assuming $E_x > E_y$, then

$$x + y = 2^{E_x}[(1.f_x)_2 + (1.f_y)_2 2^{-(E_x - E_y)}]$$

Let us assume

$$d = E_x - E_y$$

then the decimal y needs to be shifted d places to the right before participating in the addition. If the value of d is very large, then after shifting the significant digits of y to the right, many of the lower-order digits will be shifted away. If d is so large that it exceeds the range that the fraction can accommodate, the decimal part may even disappear completely, resulting in the final result of $x + y = x$ after rounding. Therefore, the mantissa bitwidth plays a crucial role in addition.

From a range perspective, addition can also overflow or underflow, but usually less drastically than multiplication. For example, adding two large positive numbers might overflow, while adding two large numbers with opposite signs might result in a very small result, potentially entering the subnormal or even underflow. Therefore, the exponent in addition still determines whether the result can be fully included. But more importantly, the difference in exponents, d, in addition directly affects the loss of precision.

In conclusion, we can clearly see that multiplication and addition have different requirements for bitwidth. Multiplication prioritizes the exponent range, but can sometimes be more lenient with the fraction bitwidth, while addition, especially accumulation, places greater emphasis on the fraction bitwidth. In multiplication, the exponent is calculated by addition, which can easily cause the result to be magnified or reduced by an order of magnitude. Its error is often a relative error, which is generally acceptable, and many applications allow for lower precision in multiplication. However, if the exponent bitwidth is insufficient, overflow or underflow can easily occur. As for addition, the precision of addition depends more on the fraction bitwidth than that of multiplication. This is mainly because addition requires alignment before it can be performed, a process that inherently results in bit loss. Furthermore, while the error in a single product is not significant during accumulation, the accumulation of numerous products leads to a substantial accumulation of errors, ultimately causing a large discrepancy between the result and the expected outcome.

2.4 Mixed-Precision

Building upon the observation that floating-point multiplication and addition exhibit fundamentally different sensitivities to precision, a natural design paradigm that emerges is mixed-precision computation. Instead of uniformly applying a single precision format across all operations, mixed-precision schemes assign different numerical formats to different stages of computation according to their sensitivity to rounding errors and dynamic range requirements.

A representative and widely adopted pattern is the combination of low-precision multiplication with high-precision accumulation [2], commonly found in dot-product and matrix multiplication workloads. For instance, operands may be represented in formats such as IEEE FP16 or Bfloat16, while the partial sums are accumulated in higher precision formats such as IEEE FP32. This design mitigates the accumulation of rounding errors and reduces the risk of information loss when adding values of significantly different magnitudes, while still benefiting from the reduced hardware cost and increased throughput of low-precision multipliers [28].

Such mixed-precision strategies are particularly well-suited to artificial intelligence workloads, especially in deep neural networks, where computations are dominated by large-scale MAC operations. Empirical studies have shown that neural networks are inherently tolerant to moderate numerical noise, as the training process can often compensate for quantization and rounding errors [29]. Consequently, reducing precision in multiplications introduces only limited degradation in model accuracy,

whereas maintaining higher precision in accumulation helps preserve numerical stability and convergence behavior. Among them, Bfloat16 is particularly favored because it has the same exponent bitwidth as IEEE FP32 so it is easy to convert [30], which can greatly reduce hardware complexity.

The advantages of mixed-precision computation are therefore multifold. From a hardware perspective, reducing operand precision significantly lowers resource utilization, power consumption, and critical path delay, particularly for multiplier units whose complexity scales super-linearly with operand width. From a system perspective, it enables higher computational density and memory bandwidth efficiency, as more data can be stored and transferred within the same bit budget. At the same time, selectively retaining higher precision in critical operations ensures that the overall numerical quality remains acceptable for target applications. This balance between efficiency and accuracy has made mixed-precision computation a cornerstone technique in modern high-performance computing and AI accelerator design.

2.5 FPGA Resources

Modern digital circuits, particularly those implemented on FPGAs, are composed of several fundamental types of hardware resources, each serving a distinct role in computation and storage. The primary resources include look-up tables (LUTs), flip-flops (FFs), digital signal processing (DSP) blocks, and on-chip memory such as block RAM (BRAM).

In Xilinx FPGAs, there exists a special structure that is not present in general digital chips: configurable logic blocks (CLB), which are used to implement sequential and combinational logic circuits. In Altera FPGAs, the corresponding structure is called as Logic Array Block (LAB). The CLB is the most abundant resources in an FPGA and consists of two SLICEs, each further divided into a SLICEL and a SLICEM, where L stands for logic and M stands for memory [31]. Therefore, CLBs can be classified into two types: CLBLL and CLBLM. The main difference is that the SLICEM contains logic capable of reorganizing LUT resources into RAM or ROM, which is the so-called Distributed RAM, while the SLICEL does not have this function. Thus, the SLICEM has additional memory and shift functionality compared to the SLICEL. Usually, a CLB contains two SLICEs, either two SLICELs or one SLICEL with one SLICEM. In Xilinx 7 series FPGA, both SLICEL and SLICEM contain four 6-input look-up tables (LUT6), three data selectors (MUX), one carry chain, and eight FFs [31]. Among these, the LUTs and FFs are the most important resources, which directly reflect the complexity of the hardware logic design.

2.5.1 Look-up Table

LUT is the core logic resource of an FPGA, essentially a small truth table memory. To be more specific, it's a 6-input, 64-depth ROM. By storing all results internally and looking them up using address lines constructed from the inputs, 6-input function logic is implemented. It is important to note that the lookup table in SLICEM

has both read and write functions, which transforms its internal LUT from a ROM into a RAM. This is also the reason why it implements shift registers and distributed DRAM functions. Although SLICEL and SLICEM have the same basic structure, they differ slightly in their more detailed structures. This results in LUTs being usable as logic function and read-only memory (ROM) in both SLICEL and SLICEM, but only in SLICEM can they be used as distributed random access memory (RAM) and shift register lookup (SRL).

2.5.2 Flip-Flop

Flip-flops are storage units within a slice and are also the basic units of sequential circuits. Each slice contains eight flip-flops, used to store data at the clock edge. It is important to note that these eight flip-flops can be divided into two main categories: four that can only be configured as edge-sensitive D flip-flops and four that can be configured as both edge-sensitive D flip-flops and level-sensitive latches. In sequential circuits, flip-flops can be used to store data and form pipelines. They can also be used to interrupt combinational logic paths, achieving timing isolation and thus improving timing. Finally, flip-flops are also a core logic resource for constructing finite state machines (FSM).

2.5.3 Dedicated DSP Blocks

Dedicated DSP blocks are specialized computational resources embedded in modern FPGAs to efficiently implement arithmetic-intensive operations. Unlike general-purpose logic constructed from LUTs, DSP blocks provide hardened architectures optimized for high-performance numerical computation, particularly multiplication and MAC operations. A typical DSP slice, like Xilinx DSP48 series [31], integrates several functional units, including a pre-adder, a multiplier, and a post-adder/accumulator, forming a datapath that supports operations of the form:

$$P = (A \pm B) \times C + D$$

These units are pipelined, allowing high operating frequencies by breaking long combinational paths into multiple stages. In addition, DSP blocks often support cascading between adjacent slices, enabling the construction of wide multipliers, long accumulators without routing intermediate results through general interconnect resources. When the design requires a large amount of computation, mapping arithmetic operations to DSP modules has significant advantages over using LUTs. Multipliers synthesized using LUTs show quadratic growth in resource usage with operand bitwidth, whereas DSP blocks provide fixed-cost, high-speed multiplication. Consequently, leveraging DSP resources can substantially reduce LUT utilization, improve timing closure and increase energy efficiency. However, DSP blocks are constrained in both quantity and operand width. When using and combining multiple DSPs, it may increase routing complexity and latency.

2.5.4 RAM

Memory resources in FPGAs are primarily implemented using embedded memory blocks and distributed memory structures. The most prominent form is BRAM, which consists of dedicated, coarse-grained memory arrays integrated into the FPGA fabric. In contrast, distributed RAM utilizes LUTs configured as small memory elements, offering greater flexibility at the cost of storage efficiency.

Block RAMs are typically organized as configurable dual-port memories, allowing simultaneous read and write operations on independent address ports. They support various configurations in terms of data width and depth, enabling designers to tailor memory structures for specific applications such as buffers, FIFOs, and lookup tables. Modern BRAMs also include optional output registers, enabling pipeline memory access to meet high-frequency timing requirements. From a performance perspective, BRAM provides significantly higher storage density and lower power consumption per bit compared to LUT-based memory. It is therefore well suited for large data storage, including intermediate results, coefficient tables, and streaming buffers in signal processing pipelines. However, BRAM access typically incurs at least one clock cycle of latency, which must be accounted for in pipeline design.

Distributed RAM implemented using LUTs is also known as LUTRAM, which provides fine-grained, low-latency storage. It is particularly suitable for small memories, register files, and cases where tight coupling with surrounding logic is required. However, its scalability is limited, as large memory structures would consume too much LUT resources and increase routing complexity.

In an FPGA system, an efficient memory design requires balancing BRAM and distributed RAM usage based on capacity, latency and resource constraints. In computation-intensive designs, such as floating-point operators and mixed-precision accelerators, memory organization plays a critical role in sustaining throughput and minimizing data movement overhead.

2.6 Error Evaluation Metrics

When designing custom arithmetic datapaths or deploying low-precision formats, we must evaluate the numerical inaccuracy introduced by hardware quantization and rounding. Academic research typically relies on three fundamental metrics to quantify these precision losses.

2.6.1 Absolute and Relative Error

The absolute error measures the straightforward mathematical difference between the exact real-number result (x) and the machine-computed result ($\hat{x}$), defined as $\epsilon_{abs} = |x - \hat{x}|$. While simple, absolute error is often inadequate for floating-point arithmetic because the representable values span a massive dynamic range. To address this, engineers utilize relative error, defined as $\epsilon_{rel} = \frac{|x-\hat{x}|}{|x|}$ (for $x \neq 0$). Relative error normalizes the inaccuracy against the magnitude of the true value,

making it a much more reliable metric for evaluating the stability of floating-point algorithms across varying exponent scales [26, Sec. 2.2].

2.6.2 Unit in the Last Place (ULP)

In computer arithmetic and hardware design, the most precise and widely adopted metric is the ULP [26, Sec. 2.6]. One ULP represents the actual numerical weight (or value) of the least significant bit of the mantissa for a specific floating-point format at a given exponent magnitude. It effectively measures the distance between two adjacent floating-point numbers.

Using ULP provides a hardware-centric way to bound rounding errors. For instance, the IEEE 754 standard for Correct Rounding (Round to Nearest) guarantees a maximum error bound of strictly 0.5 ULP [13], meaning the hardware output is the absolute closest representable number to the infinitely precise result. Conversely, alternative approaches like Faithful Rounding relax this strict bound to 1.0 ULP [12]. This relaxation ensures the output is one of the two floating-point numbers immediately surrounding the exact result, sacrificing half a bit of precision to eliminate the costly hardware needed for exact midpoint evaluation.

3

Method

This project employed a hardware-software co-simulation methodology to implement and evaluate custom Nfloat arithmetic architectures generated via the FloPoCo framework. The approach explored the trade-offs between hardware efficiency and numerical accuracy across multiple floating-point configurations. This evaluation was achieved through a comprehensive framework that integrated stimulus generation, RTL synthesis and implementation, cycle-accurate simulation, and automated mathematical error calculation.

3.1 Co-Simulation Flow

The experimental implementation followed a co-simulation approach which was partitioned into three phases. The first phase established the foundational components, utilizing the FloPoCo to generate parameterized VHDL arithmetic operators, while simultaneously employing Python scripts to generate customized test vectors that emulate AI workloads. The second phase involved the hardware instantiation, where the generated VHDL and Python-derived digital stimuli were integrated into the AMD/Xilinx Vivado Design Suite for synthesis, implementation and simulation. Finally, the third phase executed the verification and evaluation flow. In this phase, a Python-based reference script parsed the hardware calculation results from the Vivado simulator and benchmarked them against double-precision (FP64) mathematical truth values to evaluate the numerical divergence.

3.2 Synthesis and Implementation

To evaluate the performance, power consumption and hardware resource of the floating-point operators generated by FloPoCo, the generated operators were implemented using the Xilinx Vivado implementation flow. For a FPGA design, Vivado implementation provides a more realistic evaluation compared with pure RTL simulation or high-level estimation, because the design is mapped, placed and routed onto a specific FPGA device. Therefore, the reported resource utilization, timing results and power estimation are closer to what can be expected in an actual FPGA-based implementation.

For the timing constraint setup, we used out-of-context method. In this method, only the system clock was constrained with the target frequency, while no additional

input delay, output delay or board-level constraints were applied. This method allows each operator to be evaluated as an independent hardware block, avoiding the influence of external system-level interfaces or board-specific constraints. As a result, the comparison between different floating-point formats and operator configurations can focus on the intrinsic hardware cost and achievable operating frequency of the generated operators themselves.

For the MAC operators constructed by us manually combining FloPoCo-generated multipliers and adders, we paid special attention to the timing analysis after implementation. If the implementation result indicated that the direct connection could not satisfy the required timing at the target frequency, we would modify the design by adding pipeline register to make sure that no negative setup or hold slack remained. In this way, the evaluated MAC designs were not only functionally correct, but also timing-feasible for actual FPGA implementation.

3.3 Stimulus Generation and Evaluation Metrics

To accurately reflect the numerical behavior of AI workloads, a Python-based stimulus generator was developed to synthesize comprehensive test vectors conforming to a Gaussian distribution. This probabilistic approach ensures that the input stimuli closely emulate the statistical properties of weights and activations typically encountered in deep neural networks, thereby providing a more realistic assessment of the hardware's operation.

For the quantitative assessment of numerical accuracy, the evaluation adopted the mean relative error (MRE) as the primary metric, which is mathematically defined as:

$$\mathrm{MRE} = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{\hat{x}_i - x_i}{x_i}\right| \tag{3.1}$$

where N denotes the total number of test vectors, $\hat{x}_i$ represents the hardware-calculated Nfloat result for the i-th stimulus, and x_i is the double-precision (FP64) mathematical truth value generated by the Python reference model. Given the magnitude variations from floating-point representations, traditional absolute error metrics are inadequate. By normalizing the computational deviation against the magnitude of the truth value, MRE offers a standardized measure to evaluate the accuracy of the generated Nfloat architectures.

3.4 Hardware Output

To perform error analysis, we coded testbenches and used functional simulation to verify the bit-level behavior of the floating-point operators and to get reliable output results. In Vivado, simulation executes the HDL description of the design together with a testbench that provides input operands and observes the corresponding outputs after the operator latency. Since the floating-point operators are implemented as deterministic synchronous digital circuits, the simulated output is expected to match the result produced by the FPGA hardware under the same input vectors,

reset sequence, clocking scheme, and latency alignment, provided that the implemented design satisfies timing and no clock-domain crossing or initialization issues are present. Therefore, the simulation output should be acceptable as a reliable result consistent with the actual hardware output, and should be applicable to the analysis of computational errors.

By deploying a high-precision algorithmic reference model running in parallel with the hardware datapath, the framework theoretically guarantees that every synthesized RTL output can be strictly validated against an exact mathematical truth value. This closed-loop configuration provides the mathematical foundation required to systematically isolate and quantify the boundaries of hardware-induced numerical divergence.

4

Design

This chapter details the architectural design and physical implementation of the custom floating-point arithmetic units proposed in this thesis. The chapter is structured to sequentially present the hardware generation, physical performance assessment, and integration of the datapath.

4.1 Generated Floating-Point Operators

To generate floating-point operators that meet the needs of users' applications, the FloPoCo tool allows for the use of various custom targets and parameters during the generation process. For basic floating-point arithmetic, FloPoCo supports adders and FMA operators in the IEEE format, as well as adders, multipliers, dividers, comparators, and square root operators in the Nfloat format. During generating, FloPoCo allows users to customize exponent and mantissa lengths. For certain operator types, FloPoCo also supports using different exponent or mantissa lengths for the operator's input and output port. In addition, FloPoCo supports generating operator code for specific target frequencies, by using pipelining, and target FPGA hardware. Currently, the newest FloPoCo version 5.1 supports chip models include Kintex7, Virtex6, Zynq7000, and VirtexUltraScale. In this thesis, we set various target frequencies for different target FPGA hardware and examined the latency in cycles required for different types of operators at these frequencies. In subsequent experiments, we set the chip model to the Kintex7 series xc7k325tffg900-2 and the target frequency to 200MHz to ensure that the comparisons in the subsequent experiments were performed at the same frequency on the same hardware.

Adder

The adder operator generated by FloPoCo, whether in IEEE 754 or Nfloat format, generally has three inputs and one output. The inputs are system clock signal `clk` and two addend inputs `X` and `Y`, and the output is the sum R calculated by the adder. For both types of adders, inputs `X` and `Y` must reach stability and meet setup and hold times before the rising edge of the clock arrives. After the rising edge, the input data will directly enter the first stage of the pipeline and be passed between the internal registers. The output `R` is updated within the same clock cycle after the last stage pipeline is stable, maintaining a fixed number of delay cycles with the corresponding addend inputs `X` and `Y`.

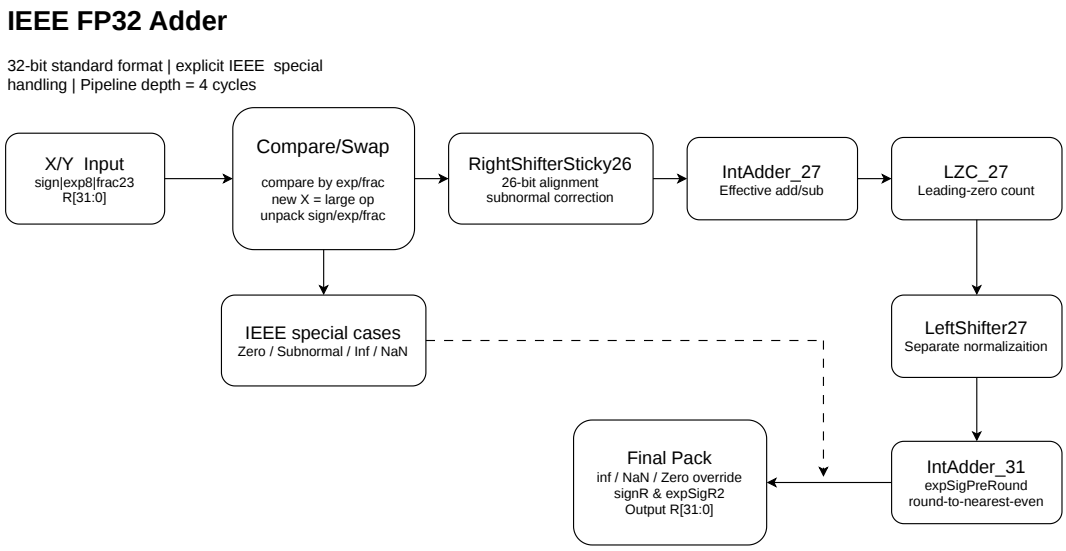


Figure 4.1: Block diagram of IEEE FP32 Adder.

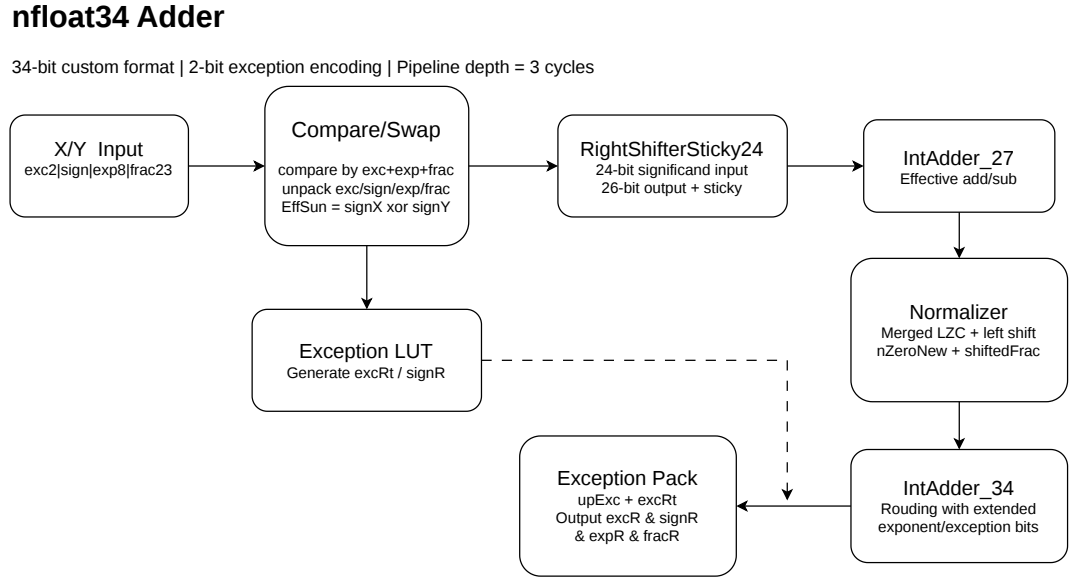


Figure 4.2: Block diagram of Nfloat34 Adder.

Figure 4.1 and Figure 4.2 demonstrate the structure of two different adders. From an internal architectural perspective, both types of adders contain several key stages required for floating-point addition. First, the `RightShifterSticky` module is used to right-shift the mantissa of the operand with the smaller exponent, aligning the exponents of the two operands while preserving the sticky bit information generated during the shift. The `FractionAdder` module performs addition or subtraction on aligned mantissa. Finally, the `RoundingAdder` module applies the rounding operation to the intermediate result. In this FloPoCo design, the rounding mode is round-to-nearest-even, which would require incrementing the last significant bit of the mantissa and may also generate a carry into the exponent field if the mantissa overflows.

The difference between the IEEE 754 adder and the Nfloat adder mainly exists in normalization stage. For the IEEE 754 format, the FloPoCo-generated adder typically contains two modules, namely `LeadingZeroCounter` and `LeftShifter_by_max`. The `LeadingZeroCounter` counts the number of leading zeros in the mantissa result and produces the required left-shift amount. The `LeftShifter_by_max` performs the actual left shift according to this amount, moving the most mantissa valid bit back to the correct position and completing the normalization step. In contrast, the Nfloat adder uses a `Normalizer` module to perform a similar function. This module combines leading-zero detection and matissa left-shifting into a single normalization stage, replacing the combination of `LeadingZeroCounter` and `LeftShifter_by_max` used in the IEEE 754 adder.

In addition, compared with IEEE 754, the Nfloat format has a more regular encoding structure for exceptional values. In Nfloat, the exception category is stored in the two most significant status bits of the floating-point word, and both the position and the width of these status bits are fixed. As a result, the operator can identify the exception categories of input and output data using relatively simple lookup-table logic, rather than decoding special values through combinations of all-zero exponents, all-one exponents and mantissa fields as required in IEEE 754. This encoding scheme simplifies the exception-classification logic and helps reduce the complexity of the associated control circuitry.

The generated VHDL code also shows how rounding is implemented in the evaluated operators. For the IEEE FP32 adder, the aligned and normalized significand is reduced to the target mantissa width before packing the final result. The generated code explicitly extracts a least significant retained bit, a round bit, and a sticky bit. The rounding increment signal is asserted when the round bit is one and either the sticky bit is also one, or the sticky bit is zero while the retained least significant bit is one. This corresponds to the round-to-nearest-even rule: values greater than half an ULP are rounded upward, while exact half-way cases are rounded toward the result with an even least significant mantissa bit. The increment is then applied by a final integer adder to the combined exponent and significand field, allowing a mantissa overflow to propagate into the exponent.

The Nfloat adder generated by FloPoCo follows the same general principle. After exponent comparison, significand alignment, addition or subtraction, and normalization, the code forms an intermediate exponent-significand vector before rounding.

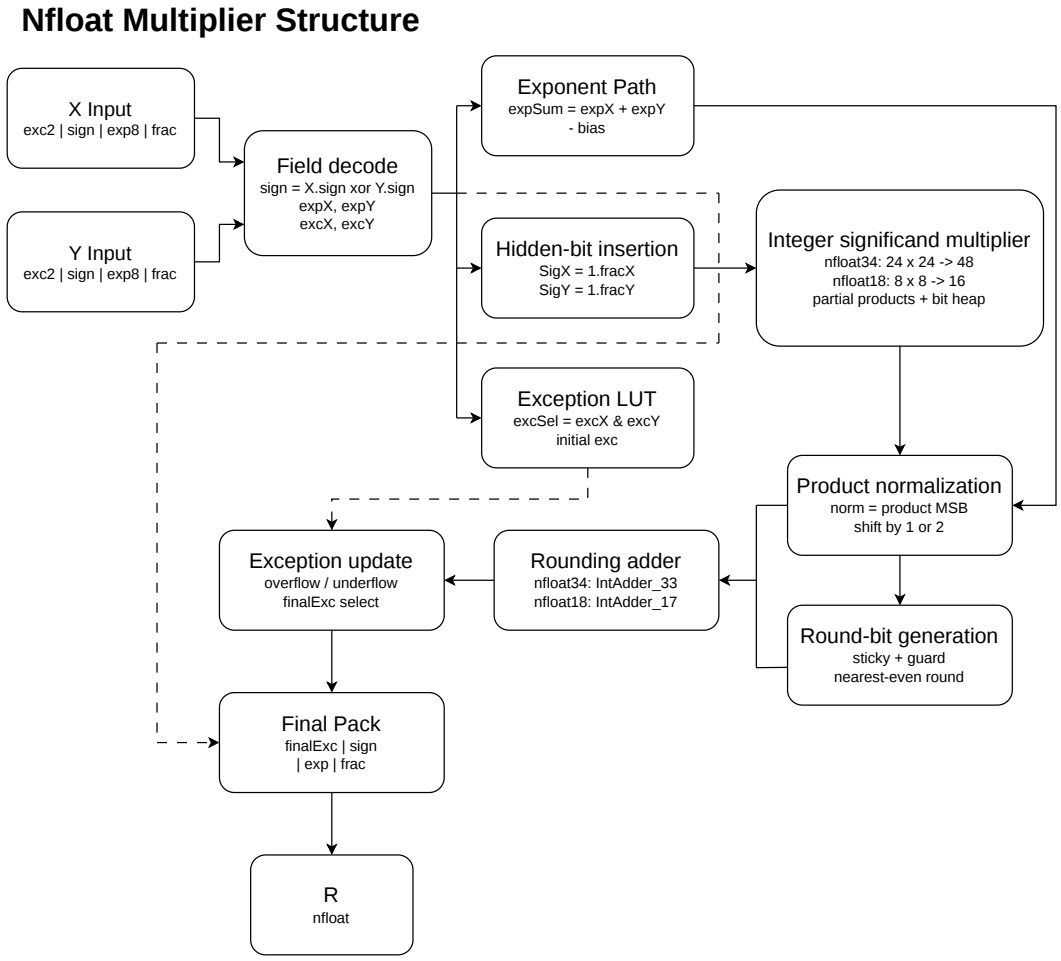


Figure 4.3: Block diagram of Nfloat Multiplier.

The signals named `rnd`, `stk`, and `lsb` are used to decide whether the result should be incremented. The condition for signal `needToRound` is asserted when the discarded part is larger than half an ULP, or when the discarded part is exactly half an ULP and the retained least significant bit is one. Therefore, although the Nfloat format differs from IEEE 754 in its explicit exception field and simplified handling of special values, the generated adder still uses a nearest-even style final rounding step for the normal arithmetic result.

Multiplier

Figure 4.3 shows the structure of Nfloat multiplier generated by FloPoCo. The multiplier mainly consists of a multiplication lookup table, a compressor, a DSP module, a small multiplier, a large multiplier, and a rounding adder. The large multiplication unit contains all the mantissas of the input with an additional 1, corresponding to the implict 1 of normal mode, is used to calculate the complete integer product of the two mantissas. When the mantissas reach a certain length, FloPoCo will use a DSP block to implement the product calculation for a computationally intensive

part, and split the remaining part into 3×3 or 2×1 blocks to supplement the low-order bits or corner overlap terms not covered by the DSP. The small block adder is implemented using LUTs, which allows for a faster computation. The compressor compresses the three inputs into two outputs, similar to the (3,2) counter in a Wallace tree. Finally, all the compressed bit stacks are added in two rows to obtain the long bit product, which is then rounded to get the final result.

The Nfloat multiplier contains an explicit rounding stage after mantissas multiplication and normalization. The two 24-bit mantissas are multiplied to produce a 48-bit product. After normalization, the retained exponent and mantissa bits are combined into an intermediate `expSig` value, while the first discarded bit and the remaining discarded bits are used to generate the rounding increment. The signal `sticky` represents the first discarded bit, while `guard` indicates whether any lower discarded bit is non-zero. Together with the least significant retained bit, these signals determine whether the `RoundingAdder` increments the result. Thus, the multiplier output is rounded before being passed to any following adder in the manually constructed Nfloat MAC.

FMA

For the IEEE 754 format, FloPoCo supports the direct generation of VHDL code for FMA operators. The basic operation implemented by this operator is $R = A \times B + C$ and it contains 6 inputs and 1 output. The input ports include the clock signal `clk`, two multiplicand inputs `A` and `B`, one addend input `C`, and two `std_logic` control signals, `negateAB` and `negateC`. The signals `negateAB` and `negateC` are used to control the signs of the product term $A \times B$ and the addend term C, respectively, during the addition stage. The only output port, R, provides the final result of the FMA operation. The block diagram of generated FMA operator is shown as Figure 4.4.

Unlike the conventional MAC structure, the FMA operator does not round the product immediately after computing $A \times B$. Instead, the unrounded product and the aligned addend C are accumulated in a wider mantissa datapath, and rounding is performed only once at the final result stage. This behavior reduces intermediate rounding error and is consistent with the fused multiply-add semantics defined in IEEE 754. From a hardware implementation perspective, the multiplication part of the FMA operator is not mainly implemented using a large amount of explicitly hand-written block multiplier logic or LUT-based structures. Instead, after the functional behavior is described, the mapping of the multiplication logic is largely determined by the synthesis tool. On Xilinx FPGAs, this logic is typically mapped primarily onto dedicated DSP48 resources.

The addition part of the FMA operator is similar to a conventional IEEE floating-point adder in some aspects, but it also has important structural differences. The main difference appears in the mantissa alignment stage. In a conventional floating-point adder, the exponents of the two input operands are compared, and the mantissa of the operand with the smaller exponent is right-shifted for alignment. However, in the FMA operator, the product term $A \times B$ is treated as the main reference, and the addend C is right-shifted to align with the exponent range of the product.

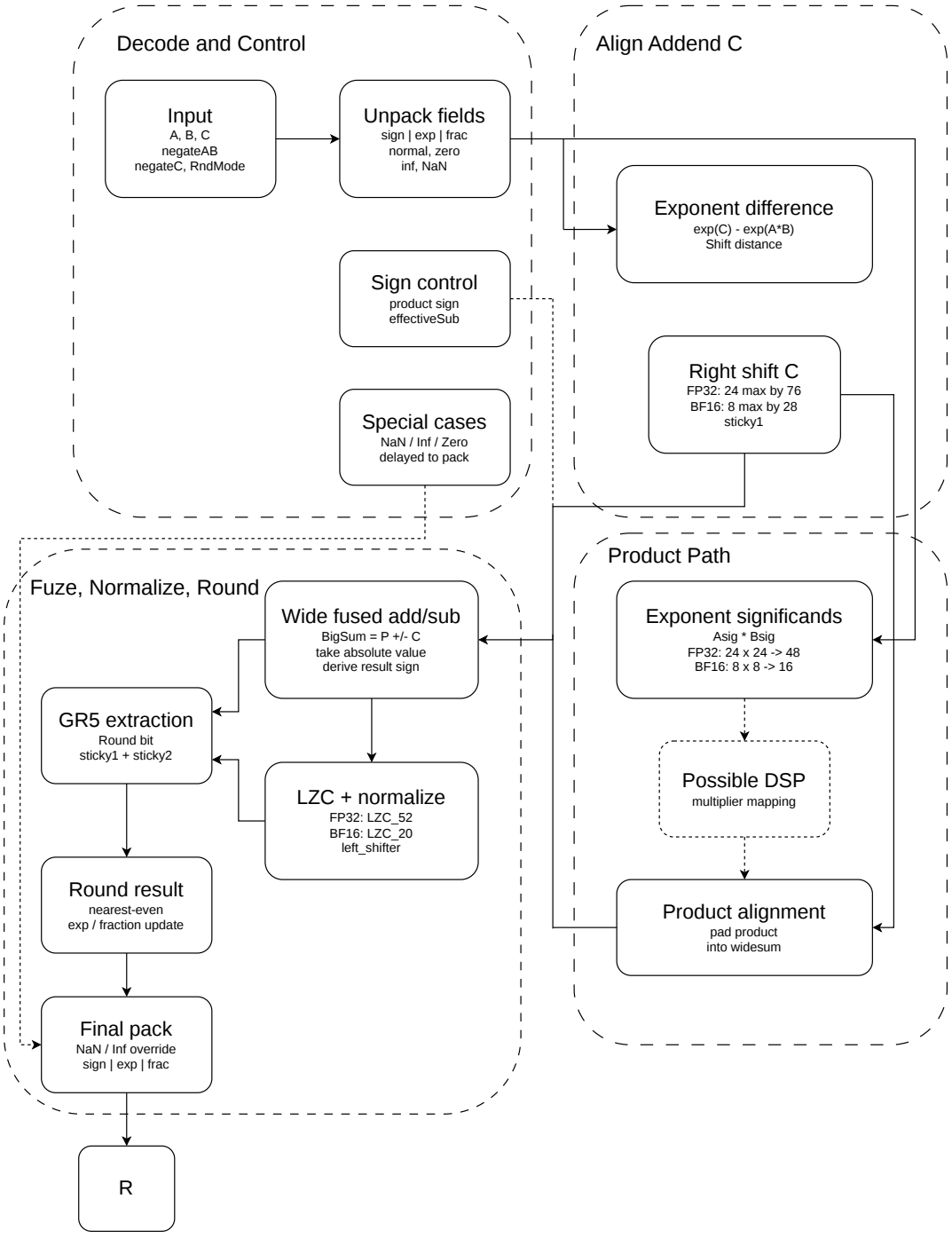


Figure 4.4: Block diagram of IEEE FMA.

Other processing stages are similar to those in a floating-point adder, such as using a leading zero counter to determine the normalization shift amount and a left-shift module to normalize the mantissa. However, because the FMA involves a multiplication result, the maximum internal mantissa width is much larger than that of a standard IEEE floating-point adder, and the maximum right-shift range required for the addend C is also greater. As a result, the internal structure and hardware cost of the FMA operator are generally higher than those of a standalone floating-point adder.

For the IEEE FP32 FMA operator, rounding is placed at the end of the fused operation rather than after the multiplication. The generated VHDL first computes the product of the two significands and aligns the addend within a wider internal datapath. Sticky information is collected both from the shifted addend and from the normalized wide summation result. Only after the multiplication and addition have been combined does the operator select the final fraction bits and compute the final round signal. The final result is then produced by adding this rounding increment to the combined exponent and fraction field. This single final rounding step is the main numerical advantage of FMA over a non-fused MAC.

MAC

Since FloPoCo does not directly support the generation of MAC or FMA operators for the Nfloat format, we constructed a top-level MAC design in this project. More specifically, the Nfloat multiplier and Nfloat adder operators were first generated separately using FloPoCo, and a custom top-level module was then written to connect and encapsulate these two operators into a complete MAC structure. The top-level MAC module contains four input ports and one output port. The input ports include the clock signal `clk`, two multiplicand inputs, and one addend input, while the output port provides the result computed by the adder. Figure 4.5 showed the structure of the MAC operator we built.

In this structure, the Nfloat multiplier first reads the two multiplicand inputs and produces the product result. The product is then connected to one input port of the Nfloat adder, while the other input port of the adder is kept as an external addend input. This input can be connected to the output of a previous MAC stage, allowing an accumulation chain to be constructed. In this way, although FloPoCo does not provide direct support for generating Nfloat MAC or FMA operators, a synthesizable and implementable Nfloat MAC module can still be built from the available Nfloat multiplier and adder operators. This module can then be used for subsequent accuracy, performance, power and area evaluations.

4.2 Utilization, Power and Timing Analysis

To evaluate the hardware cost and implementation feasibility of the generated floating-point operators, we used the synthesis and implementation flow provided by Xilinx Vivado to process the RTL codes generated by FloPoCo. In a FPGA design, the RTL code specifies the functional behavior and structural hierarchy of the

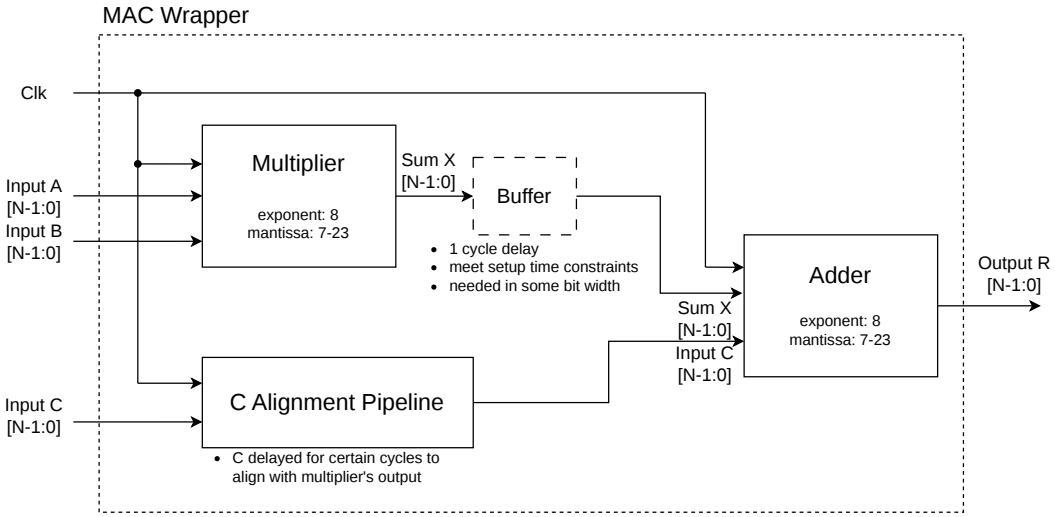


Figure 4.5: Block diagram of Unified Precision MAC.

circuit. However, the RTL code itself does not directly indicate how many FPGA resources will be consumed, how the logic will be physically placed on the device, or whether the design can operate at a given clock frequency. Therefore, synthesis and implementation are required to map the abstract RTL description onto the actual resources and physical structure of the target FPGA.

In synthesis, Vivado translates the RTL design into a technology-mapped netlist. In this process, arithmetic operations, registers, multiplexers, finite-state machines, and other logic structures described in HDL are inferred and optimized. The resulting circuit is then mapped onto FPGA primitives such as LUTs, flip-flops, carry chains, DSP blocks, block RAMs and so on. For example, combinational logic is typically mapped to LUTs, sequential logic is mapped to flip-flops, and sufficiently large arithmetic operations like multiplications may be mapped to DSP blocks. At this stage, Vivado can already provide an initial estimate of utilization, but this estimate is still based on a logical netlist rather than a fully placed and routed physical design.

The implementation process further transforms the synthesized netlist into a physically realizable FPGA design. This process includes logic optimization, placement, physical optimization and routing. In the placement step, the logical cells from the synthesized netlist are assigned to specific physical locations on the FPGA fabric. For example, LUTs and flip-flops are placed into slices, while DSP and block RAM instances are mapped to the corresponding fixed resources available on the target device. Since the physical distance between connected cells has a significant impact on routing delay, placement is a critical step for timing closure. After placement, Vivado performs routing, where the interconnections between placed cells are implemented using the programmable routing resources of the FPGA. Once routing is complete, the tool has a much more accurate view of both logic delay and interconnect delay.

Timing analysis is performed by Vivado using the timing constraints provided for the design, especially the clock period constraint. Based on the implemented netlist, Vi-

vado evaluates the delay of timing paths such as register-to-register paths, input-to-register paths, and register-to-output paths. For a synchronous register-to-register path, the data launched by a source register must propagate through the combinational logic and routing network and arrive at the destination register before the next active clock edge, while also satisfying the setup time requirement. Vivado calculates the data arrival time and the required arrival time for each constrained path, and reports the difference as timing slack. A positive setup slack indicates that the path meets the clock period requirement while a negative slack indicates a setup timing violation. In addition to setup timing analysis, hold timing will also be checked. Hold analysis ensures that newly launched data does not arrive too early at the destination register and overwrite the previously captured value. In addition, we can reduce the system clock frequency to avoid timing violations, especially setup timing violation.

The main indicators in the timing report include the worst negative slack (WNS), total negative slack (TNS), worst hold slack (WHS) and total hold slack (THS). WNS reflects the most critical setup path in the design, while TNS reflects the accumulated severity of all setup violations. Therefore, post-implementation timing analysis is used not only to determine whether the design can satisfy the target clock frequency, but also to identify critical paths and potential bottlenecks caused by long combinational logic, routing delay, high fan-out signals, or insufficient pipelining.

Power estimation in Vivado is based on the implemented design, the selected FPGA device, the clock constraints and the estimated or simulated signal switching activity. The total on-chip power is generally divided into static power and dynamic power. Static power mainly comes from device leakage and depends on the FPGA family, device size, voltage, and temperature. Dynamic power is caused by signal transitions and is related to the effective capacitance, supply voltage, operating frequency, and switching activity of the circuit. In FPGA designs, dynamic power is typically reported for different resource categories, such as clocks, signals, logic, DSP blocks, block RAMs, and I/O resources.

The accuracy of power analysis strongly depends on the quality of the switching activity information. If no simulation-based activity data is provided, Vivado will use default activity assumptions, which can provide a rough estimate but may not fully represent the actual workload of the design. More accurate power analysis can be obtained by providing switching activity generated from simulation, such as switching activity interchange format (SAIF) or value change dump (VCD) files. Nevertheless, when the same device, clock frequency, constraints, and activity assumptions are used consistently, the post-implementation power report still provides a useful basis for relative comparison between different operator implementations.

In our work, the post-implementation reports were used as the main source of performance, power, and area evaluation. This is because the implementation results take into account not only the logical structure of the generated operators, but also their mapping onto the physical FPGA fabric. This is particularly important for floating-point operators generated by FloPoCo because different exponent and fraction widths can lead to different amounts of LUT logic, DSP usage, routing complexity, and pipeline behavior. Furthermore, for composite operators such as

multiply-accumulate units constructed from separate multiplier and adder modules, post-implementation timing analysis is necessary to verify that the direct connection between submodules can satisfy setup and hold requirements at the target frequency. If timing violations are observed, additional pipeline registers can be inserted to break long paths and ensure that the evaluated design represents a practically implementable FPGA design.

4.3 Computational Error Evaluation

This section details the design for evaluating the numerical precision of the designed arithmetic units. The verification framework relies on a Python script that scales from evaluating isolated operators (by setting the accumulation length $N = 1$) to assessing deep accumulation chains.

Stimulus Generation and Filtering Strategy

For evaluating the precision of isolated arithmetic operators (e.g., standalone multipliers where $N = 1$), the testing utilizes FloPoCo's built-in test vector generator. However, these native vectors are explicitly filtered to include only 10000 normal floating-point numbers. This filtering is justified for two reasons: first, the Nfloat architecture fundamentally abandons subnormal support to maximize hardware efficiency; second, the overwhelming majority of computational values in real-world applications fall within the normal numerical range [15][17].

Conversely, the evaluation of deep accumulation pipelines (e.g., the 64-length dot product MAC) requires a data-driven testbench. As established in Section 3.3, a Gaussian distribution $\mathcal{N}(0, 0.5^2)$ is utilized. In terms of software design, the Python stimulus generator is designed to map 640000 real-valued numbers into 34-bit master Nfloat binary strings (comprising a 2-bit header, 1-bit sign, 8-bit exponent, and 23-bit mantissa). These 34-bit strings are then exported into a unified text file, functioning as the universal stimulus memory for the Vivado RTL testbench.

Quantitative Error Calculation Implementation

To realize the verification pipeline defined in Section 3.4, a error evaluation engine was implemented via a dedicated Python post-processing script. The software architecture is engineered to automate the precision assessment of the accumulation chain of MAC and FMA across variable scaling lengths (N = 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2560, 5120) utilizing a unified stimulus file. To resolve the data alignment challenges across different simulation scales, the script implements an automated chunk-based parsing mechanism. It initially loads the full sequence of paired binary stimuli into memory and decodes them into standard floats. Depending on the active hardware scaling configuration, the engine dynamically partitions the input array into continuous chunks of length N to calculate the flawless double-precision (FP64) dot-product truth values. Concurrently, the engine parses the corresponding Vivado-exported hardware log files. By mapping the paired data points through

the MRE formula, the script efficiently derives the average relative errors for each hardware scaling option.

4.4 Accumulation Chain for MAC

To evaluate the numerical error of MAC accumulation chains under different precision configurations, we coded a parameterized SystemVerilog testbench in this project to drive the FloPoCo-generated multiplier and adder submodules in simulation. The main function of this testbench is to read input test vectors from an external text file and perform multiplication and accumulation in a serial MAC manner. The input vector file is stored in plain-text format, where each line contains one binary operand pair A and B. According to the configured `N_TERMS` parameter, the testbench groups a number of consecutive operand pairs into one complete accumulation test case. For each test case, it sequentially computes the product $A_k \times B_k$ and adds it to the current accumulated value, following the operation:

$$acc_{k+1} = acc_k + A_k \times B_k$$

From the perspective of structure, the testbench uses macro definitions to select the FloPoCo multiplier and adder modules to be instantiated. This allows different operator versions, such as operators with different formats or bitwidths, to be tested without major modifications to the main testbench code. In addition, the source vector width, multiplier input bitwidth, adder and accumulator bitwidth, multiplier latency, adder latency, and the number of accumulated terms are all parameterized. During simulation, the testbench first applies the current pair of multiplicands to the multiplier input ports on the falling edge of the clock. It then waits for the configured multiplier pipeline latency and samples the multiplication result. This product is connected to one input of the adder, while the current value of the accumulation register is connected to the other input. After waiting for the configured adder latency, the testbench samples the adder output and updates it as the new accumulated value. This process is repeated until all terms in one accumulation test case have been processed.

The testbench also supports flexible combinations of different precision. The bitwidth of the source test vectors can be larger than the actual multiplier input bitwidth. In this case, the testbench truncates the operands by keeping the most significant bits, allowing multiplier with shorter significand inputs to be evaluated using the same original vector file. At the same time, when the multiplier output bitwidth is smaller than the adder bitwidth, the product is extended to the adder input bitwidth before accumulation. Therefore, this structure can be used to model mixed-precision MAC configurations, such as using a low-precision multiplier (mantissa from 7 to 22 bits) together with a higher-precision adder (mantissa at 23 bits) and accumulator. What should be noticed that this width conversion is implemented by directly preserving the higher-order bits and truncating the lower-order bits, not by a rounded format conversion.

For subsequent error analysis, the testbench writes simulation results into several output text files. On the one hand, it records the final accumulated result of each

complete test case. On the other hand, it also records the intermediate accumulated value after each accumulation step. As a result, the final-result files can be used to evaluate the overall error of a complete dot-product or accumulation chain, while the step-result files can be used to observe how numerical error propagates during the accumulation process. With this parameterized and file-driven testbench structure, different Nfloat bitwidths, multiplier-adder combinations, and mixed-precision MAC configurations can be evaluated under a unified simulation framework.

5

Results

This chapter presents the evaluation results of the custom Nfloat arithmetic architectures compared against standard IEEE 754 implementations. The evaluation is structured into three phases to thoroughly assess both physical hardware metrics and computation accuracy. First, Section 5.1 analyzes the hardware characteristics of basic operators. Section 5.2 evaluates the unified-precision MAC and FMA architectures, contrasting their resource overhead and computational error. Finally, Section 5.3 investigates the performance of mixed-precision MAC architectures and accuracy in deep accumulation chains. Unless otherwise specified, all experiments were conducted at 200MHz.

5.1 Basic Operators Analysis

This section evaluates the floating-point adders and multipliers generated by FloPoCo. Section 5.1.1 investigates the auto-pipelining behavior, comparing the required pipeline depths needed to meet timing constraints across varying target frequencies and FPGA devices. Section 5.1.2 quantifies the hardware overhead, specifically focusing on LUT consumption, DSP block allocation, and dynamic power efficiency as the mantissa bitwidth scales.

5.1.1 Pipeline Depth in different Targets

To evaluate the architectural efficiency and auto-pipelining capabilities of the generated operators, experiments were conducted across three distinct FPGA platforms: Kintex-7 (Table 5.1), Zynq-7000 (Table 5.2), and Virtex UltraScalePlus (Table 5.3). The pipeline depth of adders required to meet target clock frequencies ranging from 25 MHz to 600 MHz was recorded for four specific formats (as previously illustrated in Figure 2.1) Bfloat16, Nfloat18, standard IEEE 754 single precision (denoted here as IEEE32 for brevity), and Nfloat34.

As observed across all three tables, the pipeline depth naturally increases as the target frequency scales up, reflecting the necessity to insert more register stages to break the critical path. However, the hardware cost varies depending on the underlying FPGA architecture. The Zynq-7000 platform, representing an older generation, struggles significantly at ultra-high frequencies. For example, to achieve 600 MHz for an IEEE32 adder, the Zynq-7000 requires a 43 pipeline stages, which would incur prohibitive latency and area overhead. In contrast, the advanced Virtex

Table 5.1: Pipeline Depth vs. Target Frequency for Kintex7 Platform

Target Frequency (MHz)	**Bfloat16 (16-bit)** (via IEEEAdd)	**Nfloat18 (18-bit)** (via FPAdd)	**IEEE32 (32-bit)** (via IEEEAdd)	**Nfloat34 (34-bit)** (via FPAdd)
25	0	0	0	0
50	1	0	1	0
100	2	1	2	1
200	4	3	4	3
300	6	5	7	6
400	8	7	10	8
500	11	8	13	10
600	13	12	16	13

* Note: '0' represents purely combinatorial logic (not pipelined).

Table 5.2: Pipeline Depth vs. Target Frequency for Zynq7000 Platform

Target Frequency (MHz)	**Bfloat16 (16-bit)** (via IEEEAdd)	**Nfloat18 (18-bit)** (via FPAdd)	**IEEE32 (32-bit)** (via IEEEAdd)	**Nfloat34 (34-bit)** (via FPAdd)
25	0	0	0	0
50	1	0	1	1
100	2	2	2	2
200	5	4	6	5
300	8	6	10	8
400	12	10	14	12
500	17	14	23	20
600	28	24	43	39

Table 5.3: Pipeline Depth vs. Target Frequency for VirtexUltrascalePlus Platform

Target Frequency (MHz)	**Bfloat16 (16-bit)** (via IEEEAdd)	**Nfloat18 (18-bit)** (via FPAdd)	**IEEE32 (32-bit)** (via IEEEAdd)	**Nfloat34 (34-bit)** (via FPAdd)
25	0	0	0	0
50	0	0	0	0
100	0	0	1	0
200	1	1	2	1
300	2	2	3	2
400	3	2	4	3
500	4	3	5	4
600	5	4	7	5

UltraScalePLus platform demonstrates exceptional routing and logic performance, requiring purely combinatorial logic (0 stages) up to 50 MHz, and only 7 stages for the same IEEE32 adder at 600 MHz.

Another notable result is the comparative performance between standard IEEE operators (IEEEAdd) and FloPoCo's custom operators (FPAdd). Although Nfloat formats possess slightly larger bitwidths than their IEEE counterparts (Nfloat18 vs. Bfloat16, and Nfloat34 vs. IEEE32), the Nfloat operators consistently require fewer or equal pipeline stages across all tested frequencies and platforms.

Taking the Kintex-7 platform at 500 MHz as an example (Table 5.1), the 16-bit Bfloat16 requires 11 stages, whereas the 18-bit Nfloat18 only requires 8 stages. Similarly, the 32-bit IEEE32 demands 13 stages, while the 34-bit Nfloat34 reduces this to 10 stages. This phenomenon strongly validates the theoretical advantages discussed in Chapter 2. By explicitly utilizing a 2-bit exception field and dropping subnormal support, the FPAdd datapath avoids the complex exponent decoding, extensive normalization shifts, and sticky-bit evaluations mandated by strict IEEE 754 compliance. Consequently, Nfloat fundamentally shortens the combinatorial critical path between registers, allowing it to meet aggressive timing constraints with lower latency overhead.

While the preceding part investigated the auto-pipelining behavior of floating-point adders (FPAdd and IEEEAdd), Table 5.4 shifts the focus to floating-point multipliers (FPMult). The pipeline depth required for Nfloat18 and Nfloat34 multipliers was evaluated across the same three FPGA platforms and frequency ranges.

Table 5.4: Pipeline Depth vs. Target Frequency for Floating-Point Multipliers (`FPMult`)

Target Frequency	**Kintex7**		**Zynq7000**		**VirtexUltrascalePlus**	
(MHz)	**Nfloat18**	**Nfloat34**	**Nfloat18**	**Nfloat34**	**Nfloat18**	**Nfloat34**
25	0	0	0	0	0	0
50	0	0	0	0	0	0
100	0	0	0	0	0	0
200	0	0	0	1	0	0
300	1	1	1	1	0	0
400	1	1	2	3	0	0
500	1	2	3	6	0	0
600	2	2	8	24	1	1

* Note: '0' represents purely combinatorial logic (not pipelined).

The most obvious observation is that multipliers systematically require fewer pipeline stages compared to adders at equivalent frequencies. This architectural behavior aligns seamlessly with gate-level analyses, which demonstrate that floating-point adders inherently exhibit significantly larger logic depths (i.e., longer critical paths) than multipliers [23]. For instance, on the Kintex-7 platform at 200 MHz, both Nfloat18 and Nfloat34 multipliers remain purely combinatorial (0 stages), whereas

the adders analyzed previously required 3 to 4 stages to break these extended logic paths.

This difference roots in the datapath complexity. Floating-point addition requires extensive combinatorial logic for alignment shifting and post-addition normalization, utilizing deep barrel shifters and leading-zero anticipators. Conversely, a low-precision floating-point multiplier merely requires a small integer multiplier (e.g., 8×8 bits for Nfloat18) and a fixed maximum 1-bit right-shift for normalization, creating a substantially shorter critical path that easily meets the 5ns (200 MHz) clock constraint without pipelining.

As the mantissa bitwidth scales up, the multiplier logic shifts. While a 34-bit format naturally demands wider integer multiplication, the Nfloat34 multiplier still maintains a 0-stage pipeline at 200 MHz on the Kintex-7. At this scale, the synthesis tool automatically infers dedicated DSP slices to absorb the massive combinatorial workload. Therefore, the low latency of wide multipliers is achieved through hardware-software co-design, leveraging the FPGA's hardened silicon blocks to decrease the routing delays from a purely LUT-based wide multiplier.

5.1.2 Resources and Power

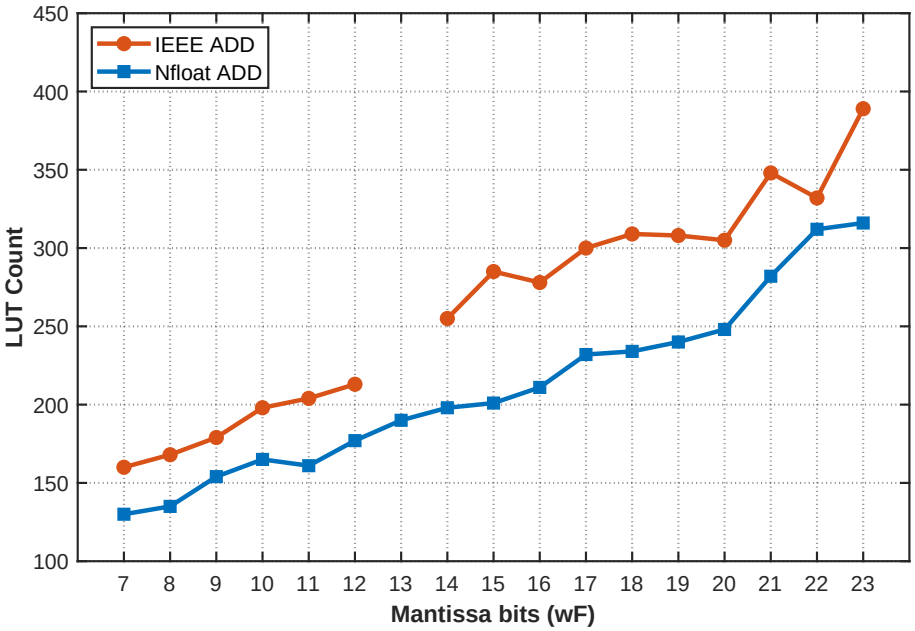


Figure 5.1: Hardware resource (LUT) comparison between IEEE ADD and Nfloat ADD architectures across different mantissa bitwidths. (Note: The data point for the IEEE ADD at $wF = 13$ is missing, as Vivado cannot synthesize HDL code at this bitwidth.)

As illustrated in Figure 5.1, the LUT consumption for both IEEE and Nfloat adders scales linearly with the mantissa bitwidth (wF ranging from 7 to 23). However, the Nfloat architecture consistently maintains a resource advantage across the entire trend. This gap visually represents the structural cost of strict IEEE compli-

ance—specifically, the logic required for subnormal detection, large normalization shifters, and sticky-bit evaluation.

This reduction in logic gates directly translates into power efficiency. According to Table 5.5, the Nfloat adder consistently consumes less dynamic power than its IEEE counterpart. At specific bitwidths, the power savings are substantial, peaking at 38.5% for $wF = 15$. Because the Nfloat datapath is streamlined with fewer toggling logic gates and shorter combinatorial paths, the overall switching activity is significantly reduced.

Table 5.5: Dynamic power comparison between IEEE and Nfloat ADD architectures across all mantissa bitwidths ($wF = 7$ to 23). (Note: Data for IEEE ADD at $wF = 13$ is unavailable.)

Mantissa Bits (wF)	**Dynamic Power (mW)**		**Power Savings (%)**
	IEEE ADD	**Nfloat ADD**	
7	7	6	14.3%
8	7	6	14.3%
9	7	7	0.0%
10	8	7	12.5%
11	8	8	0.0%
12	8	7	12.5%
13	–	8	–
14	10	8	20.0%
15	13	8	38.5%
16	12	9	25.0%
17	13	9	30.8%
18	14	9	35.7%
19	14	10	28.6%
20	14	10	28.6%
21	16	11	31.3%
22	15	12	20.0%
23	17	13	23.5%

Table 5.6 broadens the comparison to include the Posit format. While Posit has algorithmic merits, the hardware implementation results reveal severe penalties in physical design. Across all precision levels (8-bit, 16-bit, and 32-bit categories), the Posit architecture consumes more LUTs and dynamic power than both IEEE and Nfloat designs. More importantly, Posit introduces a severe latency bottleneck. For instance, the 32-bit Posit adder requires 8 clock cycles, compared to just 3 cycles for both the IEEE32 and Nfloat34 adders. This physical overhead stems from Posit's variable-length "regime" bits, which demand complex, dynamic decoding and shifting logic at runtime, making it highly inefficient for low-latency, high-throughput pipelines.

Table 5.6: Hardware Performance Comparison: Posit vs. IEEE and Custom Nfloat Formats

Data Format / Architecture	LUT	Power (mW)	Latency (clk cycles)
Posit 8_0	172	10	6
FPAdd_IEEE 16_8_7	160	7	3
FPAdd_Nfloat 18_8_7	130	6	3
Posit 16_1	355	21	7
FPAdd_IEEE 32_8_23	389	17	3
FPAdd_Nfloat 34_8_23	316	13	3
Posit 32_2	682	42	8

Figure 5.2 shifts the analysis to floating-point multipliers (FPMult). As the mantissa width increases from 7 to 15 bits, the LUT consumption rises linearly, indicating that the synthesis tool implements the multiplication entirely using general fabric logic. However, at $wF = 16$, the LUT count sharply drops, directly corresponding to the DSP allocation jumping from 0 to 1. This demonstrates that once the multiplier operand exceeds a specific threshold, the synthesis tool automatically infers a DSP block (e.g., DSP48) to absorb the bulk of the arithmetic workload. This hardware-software co-design interaction highlights how wider floating-point multipliers effectively offload computational stress from the general FPGA routing fabric into dedicated silicon primitives.

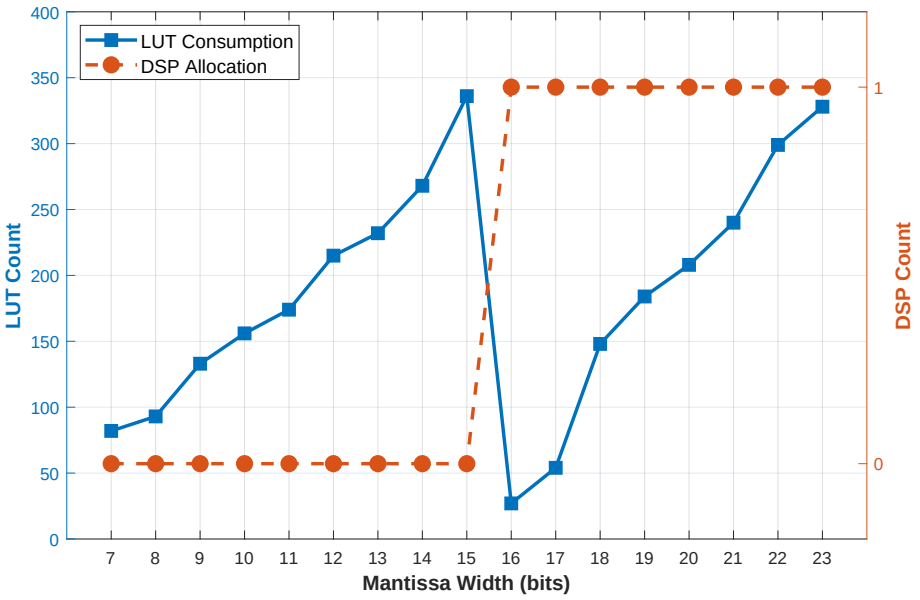


Figure 5.2: FPMult Hardware resource (LUT) Utilization across different mantissa bitwidths.

5.2 Unified-Precision Evaluation: FMA vs. MAC

This section evaluates the hardware overhead, power consumption and relative error of the custom Nfloat MAC architecture compared to the standard IEEE FMA operator across varying mantissa bitwidths ($wF = 7$ to 23).

5.2.1 Resource and Power

As illustrated in Figure 5.3, the architectural complexity of the IEEE FMA translates directly into a surge in resource consumption. Throughout the entire mantissa spectrum, the IEEE FMA consumes substantially more LUTs than the Nfloat MAC, even requiring more than double the logic resources.

The lower subplot of Figure 5.3 shows different DSP allocation thresholds dictated by the synthesis tool. The IEEE FMA demands its first DSP block very early at $wF = 9$ and forces the allocation of a second DSP block at $wF = 20$ to adapt the expanding mantissa multiplier. Conversely, the Nfloat MAC completely avoids DSP utilization until $wF = 16$.

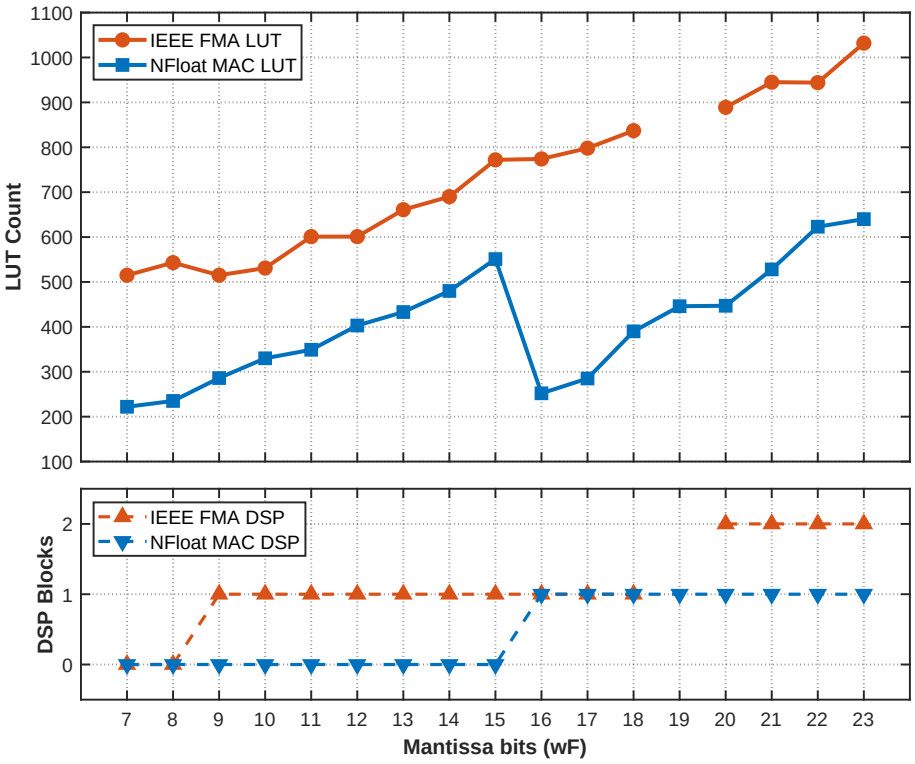


Figure 5.3: Hardware resource comparison between IEEE FMA and Nfloat MAC architectures across different mantissa bitwidths. The upper subplot shows the LUT consumption, while the lower subplot details the usage of DSP blocks. (Note: The data point for the IEEE FMA at $wF = 19$ is missing, as Vivado cannot synthesize HDL code at this bitwidth.)

A notable synthesis behavior which is a drop occurs in the Nfloat MAC curve at $wF = 16$ in LUT consumption coinciding with the allocation of its first DSP block.

This demonstrates that as the MAC width crosses the DSP inference threshold, the synthesis tool effectively absorbs the combinatorial logic into the hardened DSP macro, reducing fabric routing. The IEEE FMA does not exhibit such a prominent optimization drop, as its dominant LUT consumer is not just the multiplier, but the massive alignment and normalization shifting logic, which cannot be absorbed by a standard DSP slice.

The reduction in logic gates directly dictates the dynamic power efficiency. Table 5.7 quantifies the dynamic power consumption of both architectures. The Nfloat MAC consistently operates at a fraction of the power required by the IEEE FMA. The power savings are huge, averaging well over 50% across most bitwidths, and peaking at 63.8% at $wF = 16$ (the exact point of optimal DSP integration). For physical AI workloads where the strict single-rounding guarantee of an IEEE FMA is often algorithmic overkill, the Nfloat MAC provides an overwhelming performance-to-power ratio, making it highly advantageous for dense, parallel computing arrays.

Table 5.7: Dynamic power comparison between IEEE FMA and Nfloat MAC architectures across all mantissa bitwidths ($wF = 7$ to 23). Power savings are calculated relative to the IEEE FMA baseline. (Note: Data for IEEE FMA at $wF = 19$ is unavailable.)

Mantissa Bits (wF)	**Dynamic Power (mW)**		**Power Savings (%)**
	IEEE FMA	**Nfloat MAC**	
7	26	11	57.7%
8	28	11	60.7%
9	28	13	53.6%
10	30	15	50.0%
11	33	16	51.5%
12	37	18	51.4%
13	41	19	53.7%
14	43	21	51.2%
15	45	28	37.8%
16	47	17	63.8%
17	49	21	57.1%
18	52	24	53.8%
19	–	25	–
20	57	27	52.6%
21	60	32	46.7%
22	60	36	40.0%
23	67	39	41.8%

5.2.2 Computational Error

Figure 5.4 illustrates the MRE of the IEEE FMA and Nfloat MAC architectures on a logarithmic scale as the mantissa bitwidth (wF) scales from 7 to 23. The overarching trend for both architectures is an exponential reduction in relative error (a linear descent on the log-scale chart) as hardware bitwidth increases, validating that wider datapaths suppress quantization noise.

In the low-to-medium precision spectrum ($wF \leq 17$), the error curves of the Nfloat MAC and IEEE FMA are virtually indistinguishable. Despite the massive reduction in LUT and DSP resources, the Nfloat MAC achieves identical algorithmic fidelity to the strict IEEE standard in this range. This indicates that for typical AI workloads utilizing narrower bitwidths, the Nfloat MAC has the same theoretical accuracy advantage with the FMA.

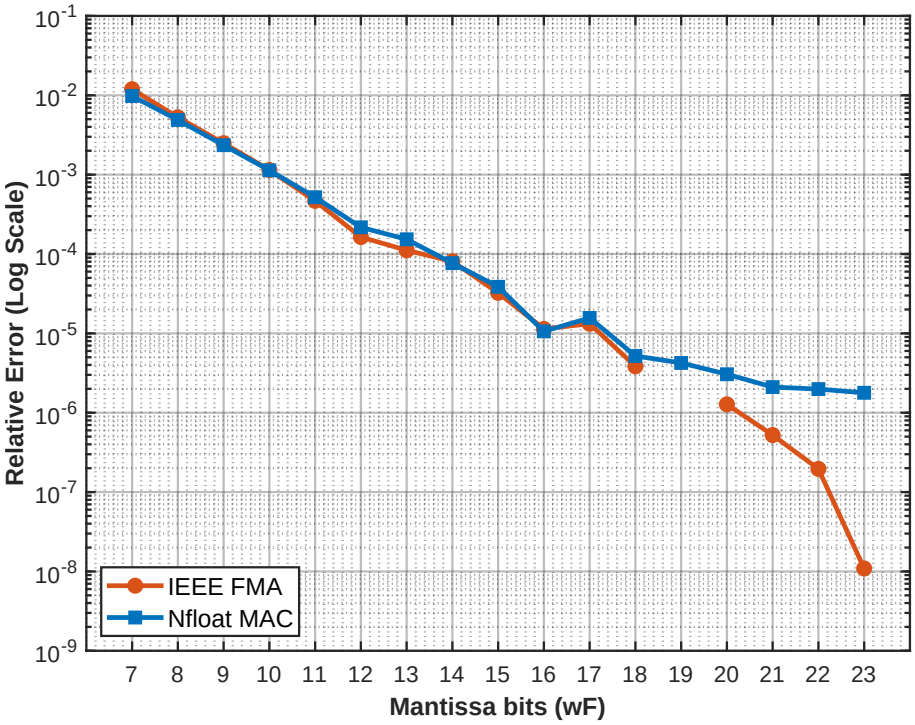


Figure 5.4: Mean relative error comparison between IEEE FMA and Nfloat MAC architectures across different mantissa bitwidths (logarithmic scale). (Note: The data point for the IEEE FMA at $wF = 19$ is missing, as Vivado cannot synthesize HDL code at this bitwidth.)

However, a distinct architectural divergence becomes apparent at higher bitwidths, particularly from $wF = 20$ to 23. While the IEEE FMA's error continues to decrease, reaching an exceptional 10^{-8} at $wF = 23$, the Nfloat MAC's error reduction begins to plateau, flattening out around the 10^{-6} mark.

The IEEE FMA calculates the exact product of $A \times B$ with with full precision and seamlessly aligns it with C before applying a single terminal rounding step. In contrast, a standard MAC architecture suffers from an intermediate rounding step immediately after the multiplication and prior to the addition. At lower precisions

($wF \leq 17$), the intrinsic quantization error of the floating-point format dominates the system, effectively masking this intermediate rounding loss. At higher precisions ($wF \geq 20$), the intrinsic quantization error shrinks, exposing the MAC's intermediate rounding error as the dominant accuracy bottleneck, thereby preventing the curve from reaching the FMA's theoretical minimum.

Despite this divergence at high bitwidths, the Nfloat MAC maintains a highly robust mean relative error of $\approx 10^{-6}$. Given that deep neural network models rarely demand strict single-precision (IEEE32) accuracy, and typically achieve state-of-the-art performance using significantly reduced precision formats (e.g., 16-bit or 8-bit) without task-level degradation [2],[16], this microscopic loss in accuracy is a favorable architectural trade-off considering the 50%+ dynamic power and area savings demonstrated previously.

5.3 Mixed-Precision MAC and Accumulation Chain

This section evaluates the hardware resource consumption and accuracy of the proposed mixed-precision MAC compared to the standard unified-precision MAC. In a mixed-precision architecture, the internal multiplier operates at a reduced target precision (mantissa width wF ranging from 7 to 22), while the subsequent accumulator and its associated alignment shifters are fixed to a high precision (typically 23 bits, equivalent to standard IEEE single precision).

5.3.1 Resource

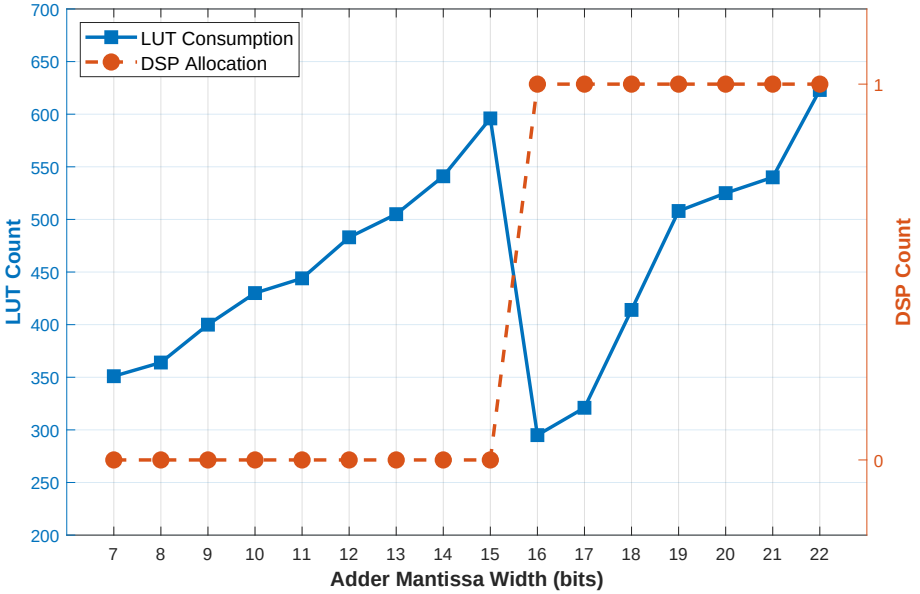


Figure 5.5: Mixed-Precision MAC Resource Utilization vs Adder Mantissa Width across different mantissa bitwidths.

Figure 5.5 illustrates the LUT consumption and DSP allocation for the mixed MAC as the multiplier mantissa width scales. The resource trend exhibits a piecewise

linear behavior, influenced by the synthesis tool's DSP inference logic. From $wF = 7$ to 15, the LUT count increases linearly as the entire MAC datapath is mapped to general fabric logic.

An architectural inflection point occurs at $wF = 16$. The DSP block allocation jumps from 0 to 1, accompanied by a drop in LUT consumption (from nearly 600 down to approximately 300). This confirms that, similar to the standard MAC, the mixed-precision architecture effectively offloads the expanding multiplication workload into DSP slices once the operand width crosses the inference threshold, leaving the LUTs to predominantly handle the fixed-width high-precision accumulation datapath.

Table 5.8: LUT Consumption and Overhead: Unified-Precision vs. Mixed-Precision MAC Architectures

Precision Format	**LUT Resource Comparison**		
	Unified MAC	**Mixed MAC**	**LUT Overhead (%)**
18_8_7	222	351	+58.1%
19_8_8	235	364	+54.9%
20_8_9	286	400	+39.9%
21_8_10	330	430	+30.3%
22_8_11	349	444	+27.2%
23_8_12	403	483	+19.9%
24_8_13	433	505	+16.6%
25_8_14	480	541	+12.7%
26_8_15	551	596	+8.2%
— Architectural Shift: Utilization of 1 DSP block begins here —			
27_8_16	252	295	+17.1%
28_8_17	285	321	+12.6%
29_8_18	390	414	+6.2%
30_8_19	446	508	+13.9%
31_8_20	447	525	+17.4%
32_8_21	528	540	+2.3%
33_8_22	623	623	0.0%
34_8_23	640	-	-

* Overhead = (Mixed - Standard) / Standard × 100%.

* Note: '-' indicates data not available.

Table 5.8 provides a comparative analysis of the LUT overhead introduced by the mixed-precision approach. At very low bitwidths (e.g., format 18_8_7), the mixed-precision MAC incurs a massive LUT overhead of 58.1% compared to its standard counterpart. This severe penalty occurs because while the standard MAC utilizes a narrow, lightweight adder corresponding to its 7-bit multiplier, the mixed MAC

enforces a full 23-bit wide accumulator. Consequently, the massive alignment shifters and wide adders completely dominate the area cost.

As the multiplier mantissa width wF increases, the baseline standard MAC is forced to widen its own accumulator. Therefore, the relative architectural difference between the two designs narrows. By $wF = 15$, the overhead shrinks significantly to just 8.2%.

Beyond the DSP inference boundary ($wF \geq 16$), the multiplier logic is absorbed by the DSP block for both architectures. The remaining LUT disparity is dictated by the difference in adder widths. As the input precision approaches the fixed accumulator precision, the overhead diminishes entirely, culminating in a 0.0% difference at $wF = 22$.

5.3.2 Computational Error Across Variable Accumulation Lengths

To evaluate the algorithmic robustness of the proposed architectures under AI workloads, this section analyzes the cumulative error behavior as the accumulation chain length (N) scales from 4 to 5120. The evaluation contrasts the standard unified-precision MACs against the Mixed-Precision MACs. We selected four formats (Nfloat18, Nfloat24, Nfloat28, and Nfloat32 from all the results shown in Figure 5.6).

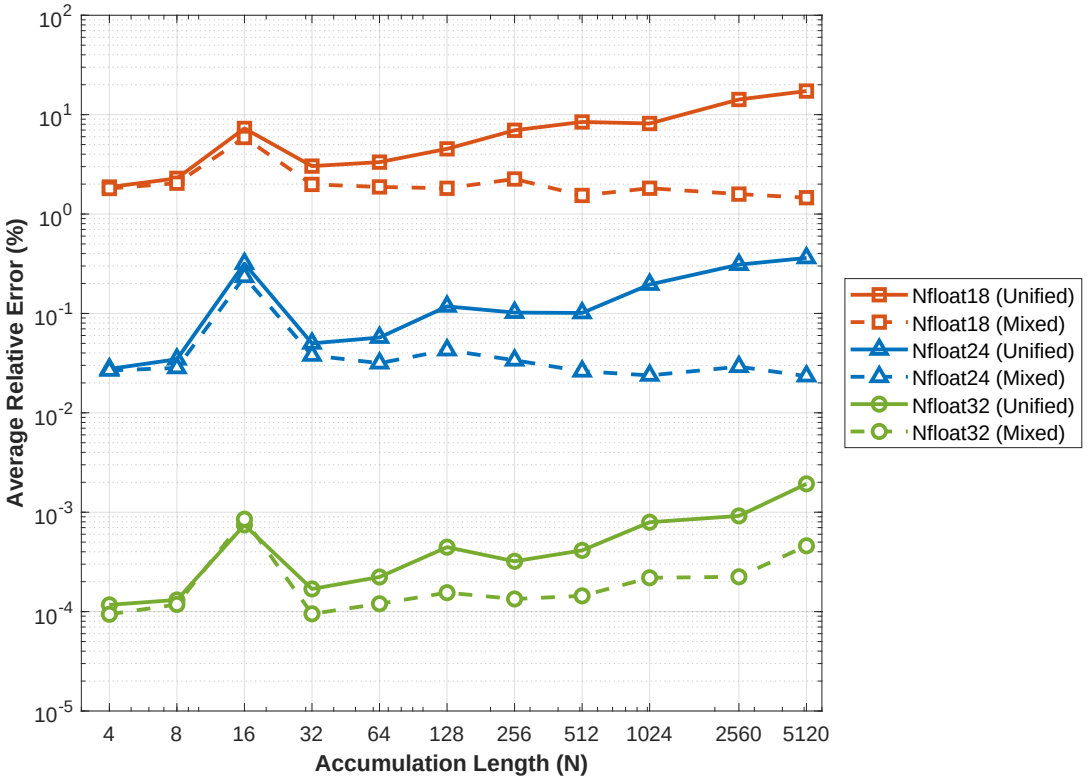


Figure 5.6: Mean relative error vs. accumulation length for unified and mixed-precision MAC

As depicted by the solid lines, the mean relative error (%) of all unified-precision

MAC upgrades significantly as the accumulation length increases. The most obvious increase is in the "Nfloat18 Unifie" curve. At $N = 5120$, the relative error of the uniform Nfloat18 MAC exceeds to over 10%, making the computational output unusable for neural network inference.

In a unified-precision MAC, the internal accumulator is restricted to the same narrow mantissa bitwidth as the multiplier. As the running sum grows larger with each cycle, the newly computed small products are shifted out of the mantissa's representable window and forcefully truncated before addition. Over thousands of cycles, these microscopic intermediate truncations accumulate, leading to severe algorithmic drift.

On the contrary, the dashed lines representing the mixed-precision MACs exhibit a flat and stable error trajectory. Regardless of the underlying multiplier precision (from Nfloat18 to Nfloat32), the error of mixed-precision architectures can be kept at a stable level. By employing a FP32 accumulator to catch the outputs of narrower multipliers, the architecture effectively acts as a lossless buffer. It absorbs thousands of accumulation operations without discarding the least significant bits of the incoming products, thereby preserving the algorithmic integrity.

6

Discussion

This chapter discusses the advantages and disadvantages of FloPoCo as a tool, the limitations of this study, and future work, finally addressing ethical issues.

6.1 FloPoCo

The experimental results of this project show that FloPoCo is a highly valuable tool for generating floating-point hardware operators. It can automatically generate VHDL code for floating-point operators according to the target frequency, FPGA device, and the required exponent and mantissa bitwidths specified by the user. Compared with manually designing complex floating-point adders, multipliers, or FMA operators, FloPoCo significantly reduces the hardware design effort and development time, while also lowering the risk of errors introduced by hand-written code. Therefore, FloPoCo is useful not only for FPGA prototyping and rapid design-space exploration, but also as a reference for operator architecture design in the front-end stage of ASIC development.

In addition to generating synthesizable hardware description code, FloPoCo is also able to automatically generate test vectors for the generated floating-point operators. These test vectors cover a variety of input scenarios, including normal numerical cases as well as some boundary or special cases, which helps verify the functional correctness of floating-point operators. Even when the generated operator code itself is not directly used, this test-vector generation capability can still be helpful for the verification of floating-point units in FPGA or ASIC designs. For floating-point hardware design, constructing representative and sufficiently diverse test inputs is often time-consuming and prone to missing corner cases, and FloPoCo's automated test-vector generation can partially reduce this burden.

However, FloPoCo still has several limitations. First, not all operator types are fully supported for all floating-point formats. For example, under the environment and requirements of this project, FloPoCo did not directly provide support for generating FMA or MAC operators in the Nfloat format, and the support for certain standalone operator types in the IEEE 754 format was also limited. As a result, this project had to construct the Nfloat MAC manually by separately generating the Nfloat multiplier and adder and then writing a custom top-level module to connect them. Second, a small number of generated operators may encounter issues during synthesis or implementation, and in some cases may fail to synthesize successfully.

Third, the architectural optimizations within FloPoCo are primarily focused on FPGAs. Consequently, when an ASIC is the target platform, the pipelined VHDL code generated by FloPoCo is not optimal in terms of area and timing efficiency. This indicates that, although FloPoCo can greatly improve the efficiency of floating-point hardware design, the generated code still needs to be verified through simulation, synthesis, and implementation before it can be considered reliable on the target hardware.

6.2 Limitations and Future Work

There are also several limitations in our own work. First, the performance, power, and area evaluation in this project was mainly conducted on a single FPGA platform and at a single target frequency. Specifically, the experiments were performed using a Kintex-7 FPGA with the target frequency set to 200 MHz. Although this setup ensures consistency when comparing different operators, it also limits the generality of the conclusions. Different FPGA devices, process generations, resource architectures, and target frequencies may all affect resource utilization, power estimation, and timing closure results. For example, at a lower frequency, some operators may require fewer pipeline stages, while at a higher frequency, additional pipeline registers may be inserted, increasing both latency and register resource usage. Therefore, future work could extend the evaluation to more FPGA targets and more frequency points in order to obtain a more comprehensive analysis of the trade-offs among performance, power, and area.

Second, this project did not further investigate why FloPoCo sometimes significantly increases the pipeline depth at relatively high target frequencies. During the experiments, it was observed that the pipeline depth of some operators does not always increase smoothly or linearly as the target frequency increases, but may instead show large jumps. This phenomenon directly affects operator latency and can further influence the overall performance of MAC chains or more complex computational structures. A deeper analysis of this behavior, such as studying the changes in critical paths at different frequencies, the internal pipeline partitioning strategy used by FloPoCo, and the bottlenecks reported in post-implementation timing reports, would help identify more suitable operating frequencies. This could avoid unnecessary latency caused by overly aggressive frequency targets and lead to a better balance among resource usage, operating frequency, and latency.

6.3 Ethical Considerations

During the preparation of this thesis, we utilized artificial intelligence tools, including ChatGPT, Gemini, to enhance the overall presentation and readability of the thesis. The application of these tools was strictly limited to language editing, grammatical correction, and the structural refinement of technical writing. Furthermore, generative AI was employed as a programming assistant to optimize LaTeX table structure and MATLAB scripts for data visualization and experimental plots.

It is imperative to emphasize that all core intellectual contributions including the architectural design, hardware implementation, experimental methodology, and the subsequent analysis and interpretation of the results are original work.

7

Conclusion

This thesis evaluated the accuracy and hardware cost of multi-format floating-point arithmetic generated by FloPoCo on FPGA. The work focused on comparing standard IEEE 754 floating-point operators with FloPoCo's custom Nfloat format, with particular attention to resource utilization, dynamic power consumption, pipeline depth, timing feasibility, and numerical accuracy. A hardware-software co-simulation flow was developed by combining FloPoCo-generated VHDL operators, Vivado-based synthesis and implementation reports, HDL simulation outputs, and Python-based error evaluation.

The experimental results show that FloPoCo is an effective framework for generating configurable floating-point arithmetic operators for FPGA-based designs. By allowing users to specify exponent bitwidth, mantissa bitwidth, target device, and target frequency, FloPoCo enables rapid exploration of different arithmetic formats and hardware configurations. The generated operators can therefore support efficient design-space exploration for customized floating-point datapaths.

Compared with IEEE 754 operators, the Nfloat format demonstrates clear hardware efficiency advantages. Because Nfloat uses a dedicated exception field and removes subnormal-number support, its datapath is structurally simpler than that of IEEE 754. As a result, Nfloat adders generally have shorter critical paths, consume fewer LUT resources, and show lower dynamic power consumption across the tested mantissa bitwidths. These results confirm that the simplified encoding and arithmetic structure of Nfloat can reduce hardware overhead while preserving useful floating-point functionality.

The comparison between the manually constructed Nfloat MAC and the IEEE FMA further confirms the efficiency of the Nfloat-based architecture. Across most tested precision configurations, the Nfloat MAC achieves substantial reductions in LUT usage, DSP usage, and dynamic power compared with the IEEE FMA. At low-to-medium mantissa bitwidths, the numerical error of the Nfloat MAC is almost identical to that of the IEEE FMA, indicating that the simplified MAC architecture can provide comparable algorithmic fidelity for reduced-precision workloads. Although the IEEE FMA continues to achieve lower error at very high mantissa bitwidths due to its single-rounding behavior, the Nfloat MAC still maintains a low mean relative error while offering significant hardware savings.

The mixed-precision MAC evaluation demonstrates the importance of using a higher-precision accumulator in long accumulation chains. When the same low precision is

used for both multiplication and accumulation, the relative error increases significantly as the length of accumulation chain grows. In contrast, the mixed-precision MAC, which combines lower-precision multiplication with higher-precision accumulation, maintains a much more stable error level even for deep accumulation chains up to N = 5120. This result supports the suitability of mixed-precision arithmetic for AI-like workloads dominated by large numbers of MAC operations.

Overall, our thesis shows that FloPoCo-generated Nfloat arithmetic provides an efficient and customizable alternative to conventional IEEE 754 floating-point arithmetic on FPGA. For applications that can tolerate reduced precision and do not depend on subnormal-number behavior, Nfloat-based operators can achieve a favorable balance between numerical accuracy and hardware cost. In particular, the Nfloat MAC and mixed-precision MAC architectures demonstrate strong potential for FPGA-based AI and DSP accelerators, where high computational density, low power consumption, and acceptable numerical accuracy are all critical design objectives.

Bibliography

[1] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, vol. 37. Lille, France: PMLR, 2015, pp. 1737–1746.

[2] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," in *International Conference on Learning Representations (ICLR)*, Vancouver, Canada, 2018.

[3] S. W. Smith, *The scientist and engineer's guide to digital signal processing*. San Diego, CA, USA: California Technical Publishing, 1997.

[4] X. C. K. W. Chen Wu, Mingyu Wang and L. He, "Low-precision floating-point arithmetic for high-performance FPGA-based CNN acceleration," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 1, pp. 6–26, 2021.

[5] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[6] V. Leon, A. Xynos, D. Soudris, G. Lentaris, R. Domingo, A. Perez, D. Gonzalez-Arjona, I. Conway, and D. M. Codinachs, "Development of high-performance DSP algorithms on the european Rad-Hard NG-ULTRA SoC FPGA," in *2024 31st IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, Nov. 2024, p. 1–4. [Online]. Available: http://dx.doi.org/10.1109/ICECS61496.2024.10849017

[7] K. Govorkova, J. G. Pardinas, V. Loncar, V. Nguyen, S. Schmitt, M. Pizzichemi, L. Martinazzoli, and E. A. Smith, "Enabling low-latency machine learning on radiation-hard FPGAs with hls4ml," 2026. [Online]. Available: https://arxiv.org/abs/2602.15751

[8] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanović, "Yosys+nextpnr: an open source framework from verilog to bitstream for commercial FPGAs," 2019. [Online]. Available: https://arxiv.org/abs/1903.10407

[9] F. de Dinechin, C. Klein, and B. Pasca, "FloPoCo: a framework for generating application-specific floating-point architectures," in *2009 International Confer-*

ence on Field Programmable Logic and Applications. Prague, Czech Republic: IEEE, 2009, pp. 604–607.

[10] F. de Dinechin and B. Pasca, "Custom arithmetic datapath design for FPGAs using the FloPoCo core generator," in *2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers*. IEEE, 2010, pp. 1260–1264.

[11] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision math library," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 2, pp. 13–es, 2007.

[12] F. de Dinechin, *FloPoCo 5.0.git manual*, 2026, [Online]. Available: http://flopoco.org/.

[13] IEEE, "IEEE standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[14] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. San Francisco, CA, USA: IEEE, 2014, pp. 10–14.

[15] E. M. Schwarz, M. Schmookler, and S. Trong, "Hardware implementations of denormalized numbers," in *Proceedings 16th IEEE Symposium on Computer Arithmetic (ARITH)*. Santiago de Compostela, Spain: IEEE, 2003, pp. 104–111.

[16] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, Toronto, ON, Canada, 2017, pp. 1–12.

[17] Intel, *BFLOAT16 – Hardware Numerics Definition*, 2018, [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/bf16-hardware-numerics-definition-white-paper.pdf.

[18] NVIDIA, *Train with Mixed Precision User's Guide*, 2023, [Online]. Available: https://docs.nvidia.com/deeplearning/performance/pdf/Training-Mixed-Precision-User-Guide.pdf.

[19] P. Micikevicius, D. Stosic, N. Burgess, M. Cornea, P. Dubey, R. Grisenthwaite, S. Ha, A. Heinecke, P. Judd, J. Kamalu, N. Mellempudi, S. Oberman, M. Shoeybi, M. Siu, and H. Wu, "FP8 formats for deep learning," 2022. [Online]. Available: https://arxiv.org/abs/2209.05433

[20] P. Larsson-Edefors, "Energy-efficient computation of TensorFloat32 numbers on an FP32 multiplier," in *IFIP/IEEE 33rd Int. Conf. on Very Large Scale Integration (VLSI-SoC)*, 2025.

[21] J. L. Gustafson and I. Yonemoto, "Beating floating point at its own game: Posit arithmetic," in *Supercomputing frontiers: third Asian conference, SCFA 2017, Singapore, March 13-16, 2017, proceedings 3*. Springer, 2017, pp. 71–86.

[22] N. H. Weste and D. M. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. Boston, MA, USA: Pearson, 2015.

[23] P. Larsson-Edefors and E. Börjeson, "Efficacy of pipelining to reduce energy of floating-point adders and multipliers," in *IEEE Symp. Computer Arithmetic (ARITH)*, 2026.

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

[25] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Cambridge, MA, USA: Morgan Kaufmann, 2017.

[26] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*, 2nd ed. Cham, Switzerland: Birkhäuser Basel, 2018.

[27] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[28] P. Zamirai, J. Zhang, C. R. Aberger, and C. D. Sa, "Revisiting Bfloat16 training," 2021. [Online]. Available: https://arxiv.org/abs/2010.06192

[29] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," 2016. [Online]. Available: https://arxiv.org/abs/1609.07061

[30] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, "A study of Bfloat16 for deep learning training," 2019. [Online]. Available: https://arxiv.org/abs/1905.12322

[31] AMD, *7 Series FPGAs Configurable Logic Block User*, 2025, [Online]. Available: https://docs.amd.com/r/en-US/ug474_7Series_CLB/Guide-Contents.