electronics

MDPI

Article

A Fast Fixed-Point Implementation for Division, Reciprocal, Square Root and Reciprocal Square Root Based on Newton–Raphson Method

Gonzalo Gutiérrez-Ramos [1], Ramón Parra-Michel [1,*], Eduardo Romero-Aguirre [2], Alberto Rodriguez-García [3] and Rodrigo Jaramillo-Ramírez [4]

[1] Department of Electrical Engineering, Advanced Studies and Research Center (CINVESTAV), National Polytechnic Institute (IPN), Zapopan 45019, Mexico; gonzalo.gutierrez@cinvestav.mx
[2] Department of Electrical and Electronic Engineering, Sonora Institute of Technology (ITSON), Ciudad Obregon 85000, Mexico; eduardo.romero@itson.edu.mx
[3] All Core Engineering Department, Intel Corporation, Zapopan 45019, Mexico; alberto.rodriguez@intel.mx
[4] Circuify Semiconductors, Zapopan 45609, Mexico; rod.jaramillo@circuify.com
* Correspondence: ramon.parra@cinvestav.mx

Abstract

Division (DIV), reciprocal (REC), square root (SR), and reciprocal square root (RSR) are fundamental operations in digital signal processing (DSP), communication, and matrix decomposition applications. However, implementing these functions using dedicated hardware units increases area and resource utilization when multiple operations are required within the same system. This paper presents a multifunctional fixed-point architecture that supports DIV, REC, SR, and RSR operations within a unified Newton–Raphson-based framework. The proposed design employs scaling and de-scaling techniques to facilitate architectural parameterization across generic fixed-point formats, piecewise polynomial approximations for seed generation, and hardware sharing between the seed computation and Newton–Raphson stages to enhance overall computational efficiency. The architecture was described in Verilog–HDL and evaluated through FPGA and ASIC implementation flows. To demonstrate the feasibility of the design, the experimental validation and implementation scope were focused on a specific of 16 bits word-length. FPGA synthesis results show that the proposed multifunctional unit achieves operating frequencies comparable to dedicated implementations while reducing hardware cost by approximately 40% compared with separate arithmetic units. Exhaustive simulations using a 16-bits representation yield SQNR values ranging from 72.03 dB to 81.03 dB across the supported operations. Furthermore, ASIC implementation using an Intel 16 nm PDK confirms the feasibility of the proposed approach for advanced technology nodes under the verified format. These results demonstrate that the proposed architecture provides an effective trade-off among accuracy, latency, and hardware efficiency, making it well suited for high-performance fixed-point DSP accelerators.

Keywords: Newton–Raphson; fixed-point arithmetic; hardware architecture; division; reciprocal; square root; reciprocal square root; FPGA; ASIC

Academic Editor: Alexander Barkalov

Received: 26 May 2026
Revised: 25 June 2026
Accepted: 26 June 2026
Published: 2 July 2026

1. Introduction

Reciprocal (REC), Division (DIV), Square Root (SR), and Reciprocal Square Root (RSR) are fundamental operations in several digital signal processing (DSP) algorithms, such as

singular value decomposition, QR decomposition [1], and vector normalization or matrix decomposition [2–5]. With rapid advancements in hardware technology, the integration density of silicon-based devices is increasing exponentially. This enables the migration of an ever-growing set of functionalities from software to dedicated hardware implementations to achieve higher processing performance.

Furthermore, since fixed-point (FxP) implementations require shorter execution times and less silicon area than floating-point (FP) alternatives [6], FxP arithmetic is preferred for portable and high-speed DSP architectures [7–9].

There are several classes of algorithms to compute division and square root operations, such as digit recurrence and multiplicative-based methods. Digit-recurrence methods [10–13] generate one quotient digit per iteration by means of a residual recurrence scheme, which entails redundant additions, single-digit multiplications, and a quotient-digit selection function. These methods [14,15], such as the Coordinate Rotation Digital Computer (CORDIC) [16] and the Sweeney, Robertson, and Tocher (SRT) division algorithm [17], result in small units, but they require a large number of clock cycles that is proportional to the word length of the FxP format. Regarding multiplicative-based methods, iterative quadratically convergent approaches, such as the Goldschmidt, Newton–Raphson, and Chebyshev algorithms [18–20], feature the advantage of obtaining more than one bit per iteration at the cost of increasing hardware resources when compared with recursive algorithms [21,22].

However, all these aforementioned works considered only single-algorithm implementations. In contrast, many DSP algorithms require accounting for the full set of operations, implying that the total combined hardware area must be considered, even if the corresponding operators are not required to execute concurrently.

In this paper, an accelerator capable of performing the FxP DIV, REC, RSR, and SR operations is presented. Unlike previous works, which focused on dedicated hardware accelerators for individual arithmetic functions, the proposed design aims to maximize hardware resource utilization through a multifunctional architecture based on resource sharing. The objective is to implement a unified arithmetic unit capable of performing all these functions while reusing common computational blocks, arithmetic operators, and control structures among the considered operations. This approach is particularly attractive in DSP applications like communication receivers and channel estimation systems, where these arithmetic procedures are often required at different stages of the processing chain.

By combining sections (hardware blocks) of the considered operations, the proposed architecture provides important area savings when these procedures are called at different times, as usually happens in DSP implementations [23–25]. The algorithm uses a piecewise polynomial approximation to obtain the seed for both REC and RSR operations; it then performs Newton–Raphson (NR) iterations to compute the final REC or RSR value. Finally, a post-multiplication stage can be used to obtain the DIV or SR value. The processing for these operations requires between two and four clock cycles to provide an FxP result that is bit-accurate with respect to the corresponding rounded floating-point calculation.

Furthermore, the proposed architecture incorporates scaling and de-scaling mechanisms to support a generic framework adaptable to various fixed-point representations. The hardware is parameterizable through the word length and fixed-point format parameters, allowing the design to be tailored to different application requirements without structural modifications. However, while the architecture is inherently flexible, the formal experimental validation and implementation scope in this work are specifically focused on an unsigned $Q(16, 5, u)$ configuration. Based on this verified format, FPGA and ASIC implementation results are presented, demonstrating that the proposed multifunctional

architecture can be successfully synthesized as a standalone arithmetic accelerator for integration into larger digital systems.

This paper is organized as follows. Section 2 introduces the proposed algorithm. Section 3 provides a detailed discussion of its parameterizable hardware architecture. Section 4 presents performance results in terms of the signal-to-quantization-noise ratio (SQNR), evaluated under 16-bits fixed-point configuration. Section 5 reports the FPGA implementation results obtained using this specific verified format. Section 6 describes the application-specific integrated circuit (ASIC) implementation results in 16 nm technology, using the Intel 16 PDK to validate the physical feasibility of the chosen configuration. Finally, Section 7 summarizes the main conclusions of this work.

2. Algorithm

The proposed architecture computes the DIV, REC, RSR, and SR operations of FxP numbers a and x, as shown in Table 1.

Table 1. Clock cycles required per each operation supported by the proposed work.

Operation	Form of Computation	Clock Cycles
REC	$1/x$	2
DIV	$a(1/x)$	3
RSR	$1/\sqrt{x}$	3
SR	$x(1/\sqrt{x})$	4

The proposed algorithm consists of three main procedures:

1. First, the REC or RSR operation is computed:

$$y_{REC} = \frac{1}{x}, \tag{1}$$

$$y_{RSR} = \frac{1}{\sqrt{x}}, \tag{2}$$

where y_{REC} and y_{RSR} stand for the results of REC and RSR, respectively.

2. Secondly, a post-multiplication is applied to (1) or (2) to obtain DIV or SR, as follows:

$$y_{DIV} = y_{REC} \times a, \tag{3}$$

$$y_{SR} = y_{RSR} \times x, \tag{4}$$

where y_{DIV} and y_{SR} refer to the DIV and SR results, respectively.

3. Finally, the computed value is rounded to the nearest representable number, as described by:

$$y = \text{round}(y_{meth}), \tag{5}$$

where y_{meth} denotes any of the results from (3) or (4) (i.e., y_{DIV} or y_{SR}).

Let us review the proposed algorithm in detail; it is based on the NR method and inspired by [21,26]. The REC and RSR operations are performed by means of the first five steps, whereas the sixth step is additionally required when performing DIV or SR operations:

1. Scaling: The input operand represented in the selected fixed-point format is normalized and mapped into a predefined working interval $wr = [\alpha, \beta]$, enabling efficient execution of the subsequent arithmetic operations.

2. Seed computation: The scaled value is evaluated using a polynomial approximation to generate an initial estimate within the working interval. This estimate serves as the starting point for the Newton–Raphson iterations.
3. Newton–Raphson algorithm: The initial estimate obtained from the polynomial is refined through the Newton–Raphson procedure. One iteration is performed for reciprocal computation (REC), whereas two iterations are applied for reciprocal square root computation (RSR).
4. De-scaling: The refined result is transformed back from the working interval to the original fixed-point representation.
5. Rounding: The de-scaled value is rounded according to the selected fixed-point precision, producing the final arithmetic result.
6. Optional post-multiplication: An additional multiplication stage is performed when required. The reciprocal (REC) or reciprocal square root (RSR) result is multiplied by the corresponding factor, a or x, to obtain the division (DIV) or square root (SR) output, respectively.

The algorithmic steps described above are realized by dedicated functional units, which collectively form the hardware architecture depicted in Figure 1.

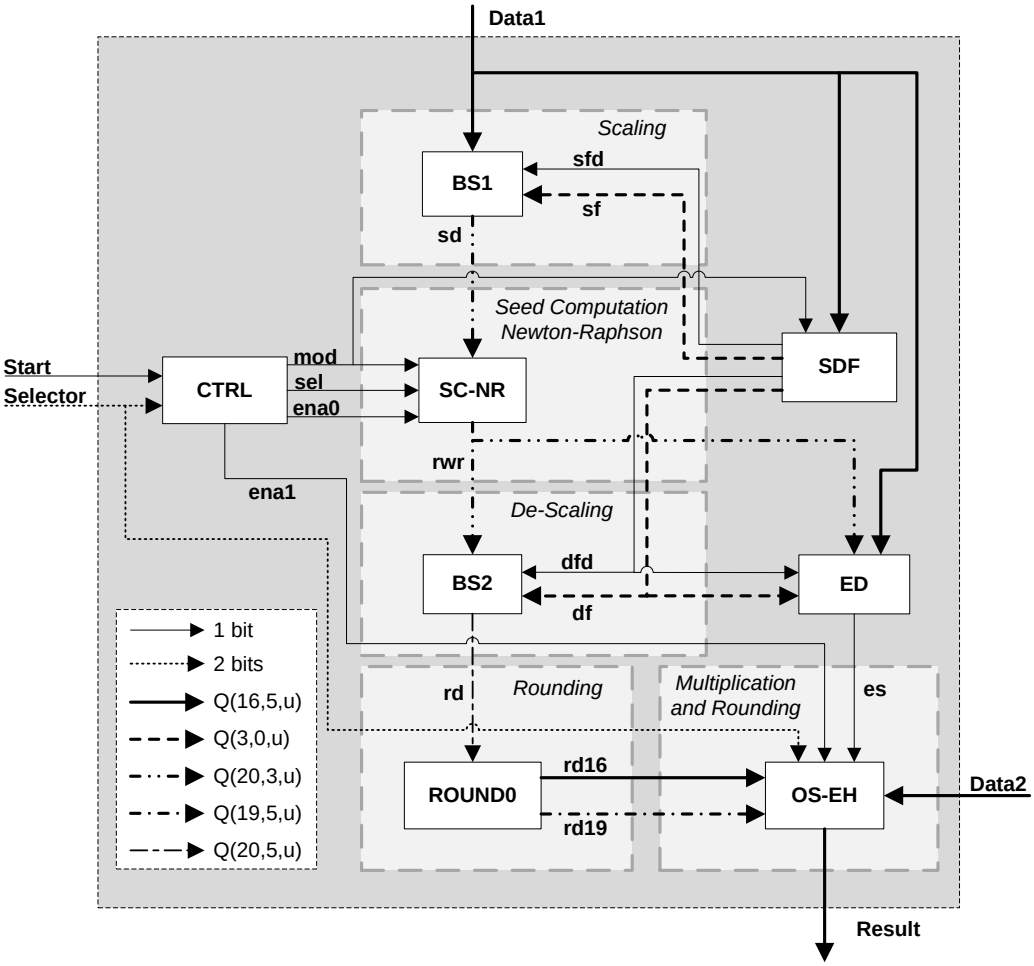

Figure 1. Proposed hardware architecture, including the word-length of each of the values involved in the process.

A value Q(·) is denoted as Q(wl, i, $sign$), where wl represents the total word length, while i corresponds to the integer bits, and $sign \in \{u, s\}$ specifies whether the format is unsigned (u) or signed (s).

Table 1 shows the operations supported by the proposed architecture, as well as the clock cycles required to perform each operation.

2.1. Scaling Step

In fixed-point implementations, arithmetic operations may require operands represented with different FxP formats, each characterized by a specific number of integer and fractional bits. Therefore, a scaling stage is necessary to develop an architecture that is independent of the underlying fixed-point representation. By mapping the input data into the working range (wr), the proposed arithmetic units can be efficiently designed for a restricted operational interval, leaving the scaling stage as the only format-dependent component of the architecture. The required scaling relationships are obtained by manipulating (1) and (2), as follows:

$$y_{REC} = \frac{1}{x} = \frac{1}{2^n}\frac{1}{\frac{x}{2^n}} = \frac{1}{2^n}\frac{1}{z}, \tag{6}$$

$$y_{RSR} = \frac{1}{\sqrt{x}} = \frac{1}{2^{n/2}\sqrt{\frac{x}{2^n}}} = \frac{1}{2^{n/2}}\frac{1}{\sqrt{z}}, \tag{7}$$

where $z = x/2^n$ and n must be chosen such that $z \in wr = [\alpha, \beta]$. It is worth noting that, in this work, $\alpha = 1$ and $\beta = 2$ for REC, while $\alpha = 0.5$ and $\beta = 2$ for RSR; these boundaries were selected to enable an efficient polynomial approximation of the corresponding operations, as shown below.

From the previous equations, it can be inferred that n must be an integer and, for (7), it must also be even. Hence, the scaling stage is implemented via a simple logical shift operation. Nevertheless, the parity requirement for n in (7) implies that the scaling step must differ for each operation. Each value of n maps an interval of possible values of x within the wr; specifically, these intervals are defined by a Lower-Bound (LB_n) and an Upper-Bound (UB_n), yielding the interval $[\mathrm{LB}_n, \mathrm{UB}_n]$ with the following concrete values:

$$\mathrm{LB}_n = \alpha 2^n, \qquad \mathrm{UB}_n = \beta(2^n) - 1/2^{frac}, \tag{8}$$

where $1/2^{frac}$ denotes the smallest fractional value representable using the chosen FxP word format, and $frac$ indicates the number of fractional bits.

2.2. Seed Computation Step

This step produces an approximate value for the selected operation within the wr. Seed computation is performed through polynomial evaluation, which provides the initial seed required for the Newton–Raphson step.

Let us first address the REC operation. To yield a high-precision result, the REC function within the wr is implemented by means of a piecewise, low-order polynomial approximation, as follows:

$$y_s = a_2 z^2 + a_1 z + a_0, \tag{9}$$

where y_s is the approximation of the result in the wr. Hence, the wr for the REC operation is divided into six segments, each represented by a second-order polynomial.

Table 2 presents the boundaries of each segment used for the REC approximation within the wr. Here, $\mathrm{LB_S}$ and $\mathrm{UB_S}$ denote, respectively, the lower and upper limits of segment S. As shown in Table 2, these values are powers of two, enabling an efficient hardware implementation of the segment selection process.

Table 2. Segment Boundaries of the Working Range (*wr*) Used in the Piecewise Polynomial Approximation for REC.

Segment S	LB_S	UB_S
1	1	1.0625
2	1.0625	1.125
3	1.125	1.25
4	1.25	1.5
5	1.5	1.75
6	1.75	2

For the RSR operation, the *wr* is divided into only two segments with boundaries $[0.5, 0.9375]$ and $[0.9375, 2]$. The *wr* is divided into two segments according to the regularity exhibited by the function over the specified interval.

Table 3 shows the FP and FxP polynomial coefficients for each segment for both REC and RSR seed computations. These coefficients were obtained using the least-squares polynomial approximation method for the desired function.

Table 3. Coefficients of the piecewise polynomial approximation in SR and RSR operations.

i	Format	*a*2	*a*1	*a*0
1 △	FP	0.9174	−2.8360	2.9187
	FxP	0x1D5B4	0xA53FA	0x5D65B
2 △	FP	0.7684	−2.5197	2.7509
	FxP	0x18972	0xAF5E9	0x58077
3 △	FP	0.6018	−2.1424	2.5375
	FxP	0x13424	0xBB718	0x51333
4 △	FP	0.3901	−1.6070	2.1993
	FxP	0x0C7BD	0xCC939	0x46609
5 △	FP	0.2354	−1.1464	1.8567
	FxP	0x07886	0xDB509	0x3B6A1
6 △	FP	0.1528	−0.8591	1.6068
	FxP	0x04E42	0xE481E	0x336B5
1 ◇	FP	0.8618	−2.0967	2.2416
	FxP	0x1B940	0xBCE80	0x47BB7
2 ◇	FP	0.1641	−0.7798	1.6157
	FxP	0x05401	0xE70BD	0x33B39

△ Segments for REC operation; ◇ Segments for RSR operation.

2.3. Newton–Raphson Step

The Newton–Raphson (NR) algorithm constitutes an efficient root-finding technique characterized by quadratic convergence. Likewise, the reciprocal (REC) operation can be computed using the iterative Newton–Raphson formulation reported in [27]:

$$w_{i+1} = w_i(2 - zw_i), \tag{10}$$

and for the RSR operation as follows:

$$w_{i+1} = \frac{w_i}{2}(3 - zw_i^2), \tag{11}$$

where w_i represents the estimate at the i-th iteration, while w_0 is the initial seed, i.e., $w_0 = y_s$. The quadratic convergence of the NR method enables the number of accurate bits to approximately double at each iteration, thereby reducing the number of required refinement steps.

2.4. De-Scaling Step

The output of the NR step is expressed within the wr; consequently, a de-scaling stage is necessary to recover the corresponding value in the original domain. The corresponding relationships can be readily derived from (6) and (7) as:

$$y_{REC} = \frac{1}{2^n} \times w_1, \tag{12}$$

$$y_{RSR} = \frac{1}{2^{n/2}} \times w_2, \tag{13}$$

which only requires a bit-shift operation of n or $n/2$ positions, resulting in a negligible hardware overhead depending on the selected operation. Note that (12) uses only one iteration of w_i from (10), while (13) uses two iterations of w_i defined in (11).

2.5. Rounding Step

To deliver the final result with the same word length as the input value, a rounding step is incorporated to eliminate the extra bits resulting from the NR method, as specified by (5).

2.6. Post-Multiplication Step

When a DIV or SR result is required, an additional post-multiplication stage must be applied to the outputs of the REC or RSR operations, respectively, as indicated in (3) and (4). The resulting value is subsequently rounded to the nearest representable number according to the selected FxP format.

3. Architecture

Figure 1 presents the architecture of the proposed multifunctional arithmetic unit. The module interface is composed of two data inputs, Data1 and Data2, a single output, Result, and two control signals, Start and Selector. The execution of an arithmetic operation begins when Start is activated, indicating that the input operands are valid and that a new computation must be performed. The Selector signal determines the operation to be executed by the accelerator and configures the internal data path accordingly.

The use of a common interface for all supported operations allows the architecture to share computational resources among multiple arithmetic functions, reducing hardware duplication and simplifying its integration into larger digital signal processing systems. Depending on the selected operation, the operands provided through Data1 and Data2

are routed to the corresponding processing blocks, while the resulting value is delivered through the Result output after the required number of clock cycles.

Table 4 summarizes the encoding of the Selector signal and specifies the role of Data1 and Data2 for each supported operation. It also indicates the corresponding functionality associated with each configuration, providing a unified view of how the proposed architecture performs division (DIV), reciprocal (REC), reciprocal square root (RSR), and square root (SR) computations using a shared hardware framework.

Table 4. Selector value and meaning of the inputs signals for selecting the operation to be performed in Figure 1.

Operation	Selector	Data1	Data2
RSR	00	Radicand	-
SR	01	Radicand	-
REC	10	Denominator	-
DIV	11	Divisor	Dividend

Following the architectural framework presented in [21], the proposed architecture is divided into the following main components: Control (CTRL), Barrel Shifter 1 (BS1), Seed Computation and Newton–Raphson (SC-NR), Barrel Shifter 2 (BS2), Rounding (ROUND0), Scaling and De-Scaling Factor (SDF), Error Detector (ED), and Operation Selector and Error Handler (OS-EH). A detailed description of each functional block is provided below.

3.1. Scaling and De-Scaling Factor Block

The SDF block generates the parameters required to perform the scaling and de-scaling operations. It receives Data1 and mod as inputs and produces the signals sf, sfd, df, and dfd as outputs. The signals sf and sfd determine the magnitude and direction (left or right) of the scaling shift operation, respectively. Similarly, df and dfd specify the magnitude and direction of the de-scaling shift. The values of sf, sfd, df, and dfd are calculated through the following two-step procedure:

- Step 1 (Position Detection): First, the position P of the most significant bit with a logic-one value is determined with respect to the binary point in the FxP representation. When mod = 1, which applies to the REC and DIV operations, it is calculated as:

$$P = MS1 - frac, \tag{14}$$

  where $frac$ is the number of fractional bits in Data1 and $MS1$ represents the index of the most significant bit in Data1 that is a logic one. Conversely, when mod = 0 for the RSR and SR operations, P is computed as follows:

$$P = \text{round}_e(MS1 - frac), \tag{15}$$

  where round_e represents rounding to the nearest even value; this adjustment is required due to the aforementioned constraints on n in (7).
- Step 2 (Scaling Factor Calculation): Second, the scaling factor is computed as follows:

$$SF = P - (int - 3), \tag{16}$$

  where int denotes the number of integer bits in Data1 and 3 represents the word length of the integer part in the internal format of the SC-NR block.

Consequently, sf represents the magnitude of SF, while sfd indicates its sign. Similarly, dfd is the sign of P, and df corresponds to the magnitude of P when mod = 1, or the magnitude of $P/2$ when mod = 0.

3.2. Barrel Shifter 1 Block

The algorithm begins by mapping the Data1 input into the wr. This task is carried out by the BS1 block through combinational shift operations. The inputs to this block are Data1, sf, and sfd, where Data1 is the operand to be scaled, sf specifies the magnitude of the shift, and sfd defines the shift direction. Consequently, the block generates the scaled data, denoted as sd, which is represented within the working range using the $Q(20, 3, u)$ FxP format. The selected working range depends on the operation being executed, as described in Section 2.

3.3. Seed Computation and Newton–Raphson Block

The SC-NR block integrates the Seed Computation and Newton–Raphson (NR) stages into a unified architecture that shares multiplier resources between both stages, as illustrated in Figure 2. This block receives the scaled data sd from the BS1 block, along with the control signals sel, ena0, and mod generated by the CTRL block. The output of the SC-NR block corresponds to the refined NR iterations within the working range, provided through the signal rwr using the $Q(20, 3, u)$ FxP format.

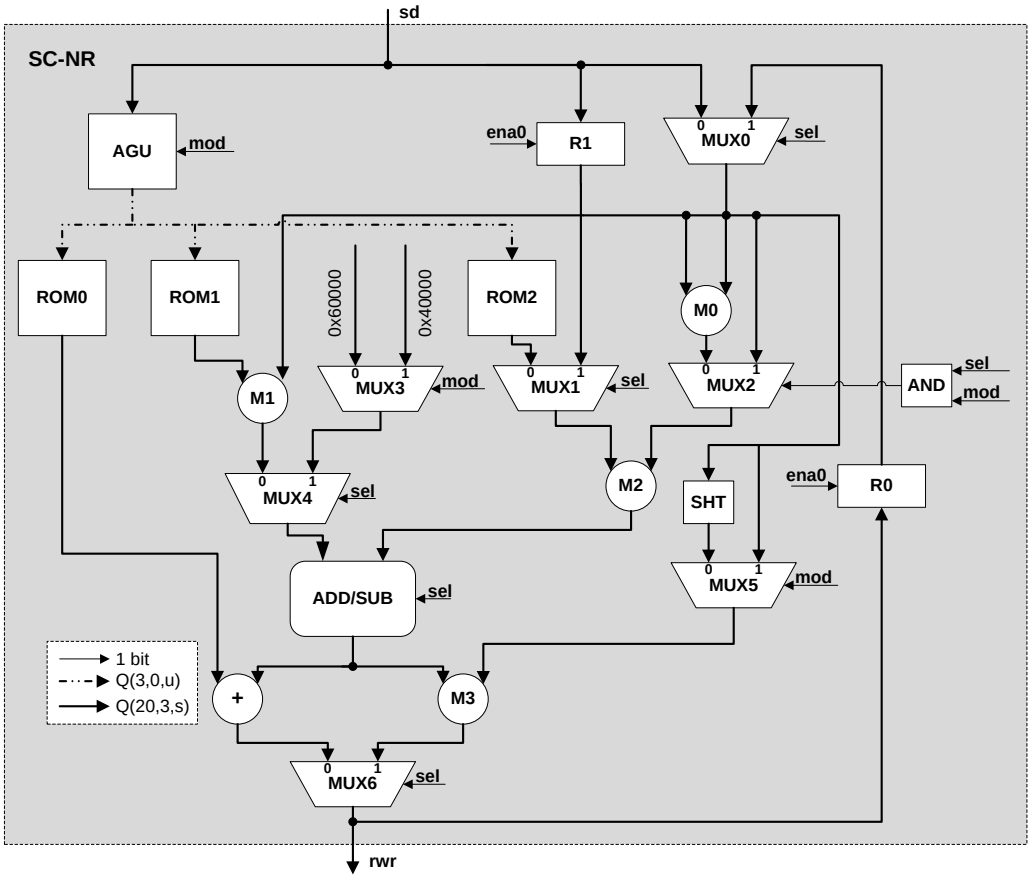

Figure 2. Architecture of the SC-NR block.

Initially, the block computes the seed value required to initialize the NR procedure. Subsequently, the NR iterations are executed, employing one iteration for the REC and DIV operations, and two iterations for the RSR and SR operations.

3.3.1. Seed Computation

The execution begins with the Seed Computation stage, which is triggered when sel = 0 and ena0 = 1. At this stage, the polynomial coefficients a_2, a_1, and a_0 used to approximate the REC and RSR results are stored in ROM0, ROM1, and ROM2, respectively; these coefficients are detailed in Table 3. Note that in this table, the first six values represent the coefficients for the REC operation, whereas the last two values correspond to the coefficients for the two segments of the RSR operation. All polynomial coefficients are represented using the $Q(20, 3, s)$ FxP format.

The signal sd serves as the input to the Address Generation Unit (AGU), which generates the address required to retrieve the appropriate polynomial coefficients from the ROMs. The generated address is defined according to the specific segment containing sd and the selected operation.

Multiplexers 0, 1, 2, 4, and 6 (MUX0, MUX1, MUX2, MUX4, and MUX6) control the dataflow to manage the NR iterations. Multiplier 0 (M0) receives the signal sd to compute the z^2 term. Multiplier 2 (M2) then multiplies the coefficient a_2, retrieved from ROM2, by the z^2 value generated by M0, yielding the product a_2z^2. In parallel, Multiplier 1 (M1) combines the coefficient a_1 from ROM1 with sd to obtain the term a_1z. The outputs of M2 and M1 are subsequently processed by the ADD/SUB block, after which the coefficient a_0 from ROM0 is incorporated. Finally, the resulting value is stored in register R0.

It should be noted that several polynomial evaluation operations are executed concurrently. In particular, the computations of a_2z^2 and a_1z are performed in parallel, reducing the overall combinational depth of the seed computation stage. Consequently, the critical path is not determined by the complete polynomial evaluation sequence, but rather by the propagation through the multiplexers, ROM accesses, multipliers, and adder/subtractor blocks involved in a single computation path. This parallelism contributes to achieving the high operating frequencies reported in Section 5.

3.3.2. NR Iteration

During the second clock cycle, with sel = 1 and ena0 = 0, the Newton–Raphson iteration is executed. The specific operation performed depends on the value of mod. Accordingly, MUX0, MUX1, MUX2, MUX4, and MUX6 select the inputs labeled with "1".

When mod = 1, corresponding to the REC mode, the seed stored in R0 is multiplied by sd in M2. Subsequently, MUX3 outputs the hexadecimal constant 0x40000, which corresponds to the value 2 represented in the $Q(20, 3, u)$ format. The ADD/SUB block, configured as a subtractor, computes the expression $2 - zw_n$. The resulting value is then multiplied in M3 by the seed stored in R0. Finally, MUX6 forwards the output of M3, yielding the REC approximation through the signal rwr in the $Q(20, 3, u)$ format.

For mod = 0, corresponding to the RSR mode, the seed previously stored in R0 is multiplied by itself in M0 and subsequently by sd in M2. In this case, MUX3 selects the hexadecimal constant 0x60000, which represents the value 3 in the $Q(20, 3, u)$ format. The ADD/SUB block, operating as a subtractor, evaluates the expression $3 - zw_n^2$. The output of this operation is multiplied in M3 by the halved seed, obtained through the logical shifter SHT. The value selected by MUX6 corresponds to the RSR approximation expressed in the $Q(20, 3, u)$ format. Since the selected operation is RSR, this refinement procedure is executed twice.

3.4. Barrel Shifter 2 Block

The de-scaling operation is carried out by the BS2 block, which employs the same barrel shifter structure asBS1. The inputs to this block are the signal rwr generated by the SC-NR block, and the signals df and dfd provided by the SDF block. The output of BS2 is

the signal rd, represented using the $Q(20, 5, u)$ FxP format. The signal rd corresponds to the de-scaled value (y') defined in (12) and (13), using a different FxP format than the input.

3.5. Rounding Block

When an FxP format is converted to another FxP format with fewer bits, a rounding process is recommended to preserve data accuracy. In this regard, the ROUND0 block performs rounding to the nearest value of the de-scaled data rd, yielding the outputs rd16 and rd19.

The signal rd16 is represented using the same FxP format as Data1 and is employed when the selected operation corresponds to REC or RSR. In contrast, rd19 is utilized for DIV and SR computations. This signal incorporates three additional fractional bits and is forwarded to the OS-EH block to perform the post-multiplication stage associated with these operations. The inclusion of these extra bits compensates for the loss of accuracy introduced during the post-multiplication step.

3.6. Error Detector Block

The ED block receives the signals Data1, rwr generated by the SC-NR block, and df and dfd provided by the SDF block as inputs. Its output is the signal es. The primary function of this block is to identify input values for which the REC or RSR results cannot be represented within the output FxP format adopted by BS2.

As an illustrative example, consider a $Q(16, 5, u)$ format where the maximum representable output value is 31.9995. When performing the REC operation, the minimum input value that avoids causing an overflow is $1/2^5$; any input below this threshold would imply the activation of the error signal es. Supporting smaller input values would require increasing the number of integer bits in the chosen FxP representation.

3.7. Operation Selector and Error Handler Block

The OS-EH block is responsible for selecting the DIV, REC, RSR, or SR operation. In addition, it determines whether the output should be replaced by a saturation value, which corresponds to the maximum representable value in the selected output format (e.g., 0xFFFF for a 16-bit configuration), whenever the computed result exceeds the representable range. The architecture of the OS-EH block is shown in Figure 3. The specific operation to be executed is determined by the input signal Selector, according to the encoding summarized in Table 4.

The multiplier unit (M4) is employed to perform the DIV and SR operations. When a DIV operation is selected, the dividend value in Data2 and the value stored in rd19 are multiplied using M4. Conversely, when an SR operation is selected, rd19 is multiplied by Data1 via M4.

The output generated by M4 is then processed by the overflow detection unit (OVFDE). If an overflow condition is detected, multiplexer 8 (MUX8) selects its input labeled “1”, thereby propagating a saturation value to multiplexer 10 (MUX10). In addition, when the es signal from the ED block is asserted, both MUX8 and MUX9 select the saturation value. In the absence of overflow and with es deasserted, the output of M4 is rounded to the nearest representable value, routed to MUX10, and finally assigned to Result.

When either the REC or RSR operation is selected, multiplexer 9 (MUX9) chooses between the signal rd16 and the saturation value. The selected value is subsequently forwarded to MUX10 and finally assigned to Result.

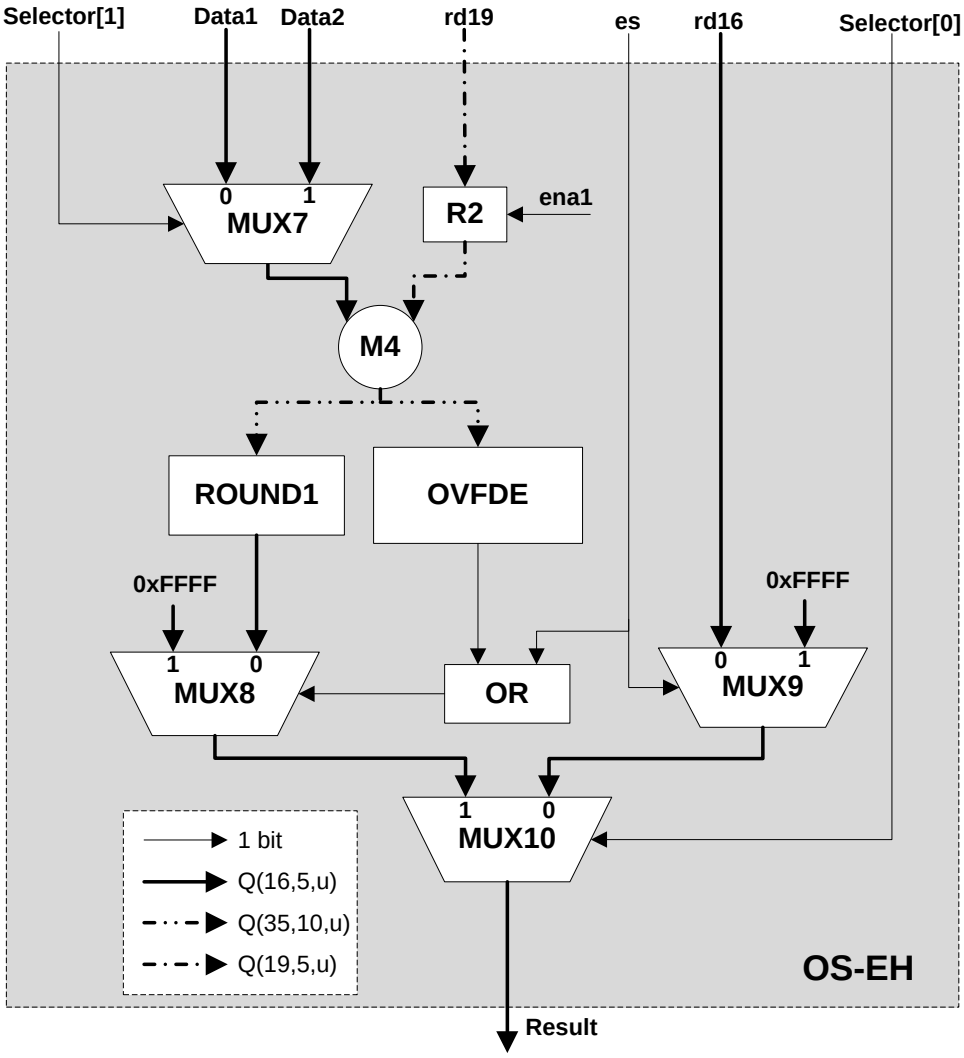

Figure 3. Hardware Architecture for the Operation Selector and Error Handler.

4. Signal-to-Quantization-Noise Ratio Analysis

The objective of the proposed algorithm is to combine four operations in only one flexible design but maintaining a good precision of the FxP result when compared to the FP results. A widely adopted metric for assessing the accuracy degradation introduced by quantization is the signal-to-quantization-noise ratio (SQNR). This metric is defined as the logarithm of the ratio between the signal power, denoted by P_s, and the quantization error power, denoted by P_e, as expressed by

$$\text{SQNR} = 10 \log_{10} \left\{ \frac{Ps}{Pe} \right\}. \tag{17}$$

The variables P_s and P_e are computed as

$$Ps = \frac{1}{N} \sum_{i=1}^{N} Vf_i^2, \tag{18}$$

$$Pe = \frac{1}{N} \sum_{i=1}^{N} (Vf_i - Vfx_i)^2, \tag{19}$$

where Vf_i and Vfx_i denote the FP and FxP results associated with the i-th experiment, respectively, and N represents the total number of experiments used in the evaluation and simulations.

To measure the Signal-to-Quantization-Noise Ratio (SQNR) of the proposed design, a comprehensive verification testbench infrastructure was developed and executed via Register-Transfer Level (RTL) simulations in Verilog-HDL within the ModelSim environment. Considering the unsigned $Q(16, 5, u)$ representation, the evaluation for unconstrained unary operations (REC, SR, and RSR) exhaustively covered all 2^{16} = 65,536 possible input vectors. For the binary division (DIV) operation, performing a full exhaustive cross-product verification would be computationally prohibitive due to the immense state space (2^{32} combinations). Therefore, a robust pseudo-exhaustive strategy was adopted: the testbench evaluated all 2^{16} = 65,536 possible values for the divisor (Data2), while a randomized set of approximately 10,000 test vectors was generated for the dividend (Data1) across each divisor configuration, yielding a total of approximately 655.36×10^6 simulation vectors. In parallel, a MATLAB version 2024b, program computed the REC, RSR, SR, and DIV reference results for the same set of inputs using 64-bit floating-point arithmetic, which were stored as the golden reference values.

The entire verification workflow was evaluated in terms of computational overhead. The net RTL simulation execution time in ModelSim required approximately 4.5 h for this extensive test suite. Subsequently, the data extraction and fixed-point to floating-point transformation within MATLAB for reference cross-examination and SQNR calculation took approximately 20 min.

Since the proposed architecture employs saturation whenever the exact result cannot be represented in the selected output format, the same criterion was applied to the floating-point golden model. Specifically, MATLAB outputs exceeding the dynamic range of the adopted FxP representation were saturated to the maximum representable value before the SQNR computation. This situation arises when evaluating corner cases such as the reciprocal of ultra-small input values near the underflow boundary or division-by-zero scenarios, whose exact floating-point results exceed the range supported by the hardware implementation. Applying the same saturation behavior ensures a fair comparison between the golden model and the proposed architecture.

All valid outputs, i.e., those representable within the selected output format after applying the aforementioned saturation criterion, were compared against the corresponding golden values using the SQNR metric expressed in dB. The obtained SQNR values for REC, RSR, SR, and DIV were 76.9852 dB, 79.2441 dB, 72.0332 dB, and 81.0289 dB, respectively. For the considered FxP format, the maximum achievable SQNR values were 76.9892 dB, 80.0153 dB, 72.0344 dB, and 89.0620 dB for REC, DIV, RSR, and SR, respectively. These results demonstrate that the proposed design provides sufficiently high SQNR performance for 16-bit DSP computations under the verified format. To support reproducibility, the simulation testbench and the MATLAB scripts used to generate the input vectors and reference results for the verification process are available from the corresponding author upon reasonable request.

5. Implementation Results

In order to show the pertinence of having a multifunctional unit versus isolated implementations for each operation, a comparison between the proposed design and separated 16-bit implementations of DIV-REC [21] and RSR-SR [26] operations is presented.

For a fair and consistent evaluation, all architectures included in this comparison were configured and synthesized using the same unsigned $Q(16, 5, u)$ fixed-point format.

The hardware designs were described in Verilog-HDL and synthesized on a Stratix-V 5SGXMA7N1F45C1 FPGA using Quartus Lite Edition version 20.1. The resulting implementation metrics are presented in Table 5.

Table 5. Comparison of synthesis results between standalone and multifunctional architectures.

Resources	[26]	[21]	Combined	This Work	Reduction
Max. Freq. (MHz)	86.31	84.93	–	85.43	–
Adaptive Logic Module (ALM)	233	286	519	320	38.34%
Dedicated Registers	64	63	127	65	48.82%
DSP Blocks	5	5	10	5	50.00%

It can be observed that the proposed architecture achieves a clock frequency comparable to those of the standalone RSR-SR and DIV-REC modules while requiring the same number of DSP blocks. Regarding ALMs and dedicated registers, the RSR-SR and DIV-REC implementations individually consume, on average, more than 80% of the hardware resources required by the proposed multifunctional RSR-SR-REC-DIV architecture. Therefore, when separate units are employed to implement each operation within the same design, the overall hardware cost nearly doubles compared with that of the multifunctional unit.

More specifically, the combined implementation of the dedicated DIV-REC and RSR-SR accelerators requires 519 ALMs, 127 registers, and 10 DSP blocks. In comparison, the proposed multifunctional architecture requires only 320 ALMs, 65 registers, and 5 DSP blocks, corresponding to resource reductions of 38.34%, 48.82%, and 50.00% for ALMs, registers, and DSP blocks, respectively. These results further highlight the effectiveness of the proposed hardware reuse methodology, which enables the execution of division, reciprocal, square-root, and reciprocal square-root operations within a unified architecture. Consequently, the proposed design significantly reduces hardware duplication while maintaining an operating frequency comparable to those of the dedicated implementations. As previously discussed, earlier studies predominantly focused on standalone implementations of arithmetic operations.

In contrast, the proposed architecture accommodates multiple operations within a single integrated hardware framework, as summarized in Table 6, thereby improving hardware utilization and reducing the overall resource requirements when several arithmetic functions are required within the same system.

Table 6. Operating algorithms supported by the compared works.

Operation	[26]	[12]	[21]	This Work
REC			X	X
DIV		X	X	X
RSR	X			X
SR	X			X

6. ASIC Implementation

The proposed architecture was implemented using a state-of-the-art Intel 16 nm technology process and the Synopsys digital implementation flow, including Design Compiler [28] for logic synthesis and IC Compiler II [29] for physical design and place-and-route. The implementation was evaluated under multiple process-voltage-temperature (PVT) cor-

ners [30], including slow, typical, and fast conditions, in order to ensure timing robustness against manufacturing variations.

The timing constraints were defined in terms of the target clock period and clock uncertainty, which were the only parameters adjusted during the implementation flow. All other constraints were kept consistent with a standard ASIC flow methodology. An iterative timing-closure strategy was employed, in which the target clock period was progressively reduced and the synthesis and physical design stages were repeated until the maximum operating frequency that still ensured timing closure was identified.

The final implementation achieves timing closure at 1.25 GHz, with positive slack observed after place-and-route across all evaluated PVT corners (slow, typical, and fast). This confirms the robustness of the design under worst-case and nominal operating conditions.

At the physical design stage, the implementation was verified under both nominal and worst-case timing scenarios, which guided floorplanning, placement, clock-tree synthesis, and routing to ensure stable operation across all conditions. The resulting post-layout netlist consists of 14,975 standard cells. Figure 4 shows the final place-and-route layout of the design.

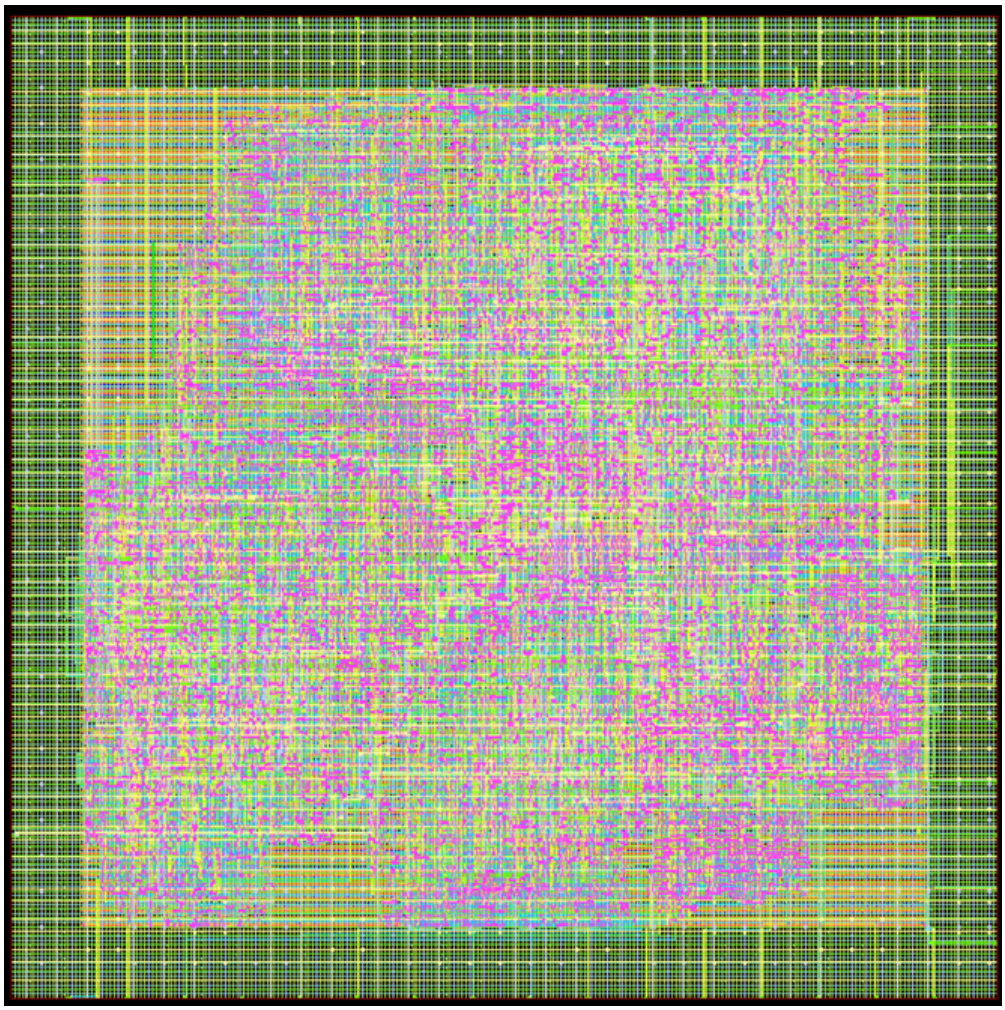

Figure 4. ASIC physical view results for Intel 16 nm PDK Technology.

To provide a fair comparison between this work and previous implementations fabricated in older technology nodes, a modern technology-scaling model based on empirical data and recent literature was applied. The original architecture presented in [31] was synthesized using a 180 nm process, whereas our proposed implementation targets a 16 nm node.

According to the updated scaling trends reported in [32], power consumption and area typically decrease by factors ranging from 6× to 10× and 50× to 200×, respectively, when migrating from 180 nm to sub-20 nm technology nodes. These empirical factors account for practical considerations in modern CMOS processes, including leakage effects, dynamic power optimization techniques, and physical design constraints, thus providing a more realistic representation of technology scaling than idealized models.

To facilitate a fair comparison, Table 7 presents three sets of results: (i) the original values reported in the reference implementation, (ii) the corresponding values normalized to a 16 nm technology node using the aforementioned scaling factors, and (iii) the actual implementation results obtained in this work using the Intel 16 nm PDK. Presenting the data in this manner provides additional insight into the relative efficiency of the proposed architecture while enabling a more representative comparison under a common technology framework.

Table 7. Comparison of synthesis results before and after technology scaling.

Metric	[31] Original	[31] Scaled	This Work Intel PDK
Technology (nm)	180	16	16
Power (mW)	40.9	6.3	6.3
Area (μm^2)	524,000	2620	278

It should be emphasized that the normalized values are intended solely as approximate estimates to aid cross-technology comparisons and should not be interpreted as actual implementation results. Consequently, only the figures obtained using the Intel 16 nm PDK correspond to measured synthesis outcomes of the proposed design.

Table 8 shows the execution time required for both compared works. The provided results confirm that te FxP implementation offers a shorter execution time than the FP implementation. Considering also that FxP implementation achieves less area resources than the FP approach, it is comprehensive that FxP format is preferred for the implementation of low-power portable devices.

Table 8. Execution time results for each operation in FP and FxP.

Operation	FP Implementation [31]	FxP Implementation (This Work)
REC	54.6 ns	1.6 ns
DIV	63.7 ns	2.4 ns
RSR	81.9 ns	2.4 ns
SR	91 ns	3.2 ns

7. Conclusions

This paper presents a multifunctional fixed-point (FxP) unit for performing DIV, REC, SR, and RSR operations. The architecture is structurally parameterizable through hardware description language parameters, enabling its adaptation to different FxP word lengths and formats. However, the comprehensive experimental validation carried out in this work was strategically scoped to an unsigned $Q(16, 5, u)$ representation to demonstrate its physical and numerical feasibility.

In terms of hardware resources, implementation results demonstrate that the proposed multifunctional unit successfully reuses hardware blocks, requiring significantly

fewer resources than separate dedicated units for each operation. Moreover, the proposed architecture computes the operations within a maximum of four clock cycles in the worst-case scenario. The physical implementation comparison shows that the proposed architecture achieves an area reduction of $\sim 10\times$ while maintaining a power consumption comparable to the reference floating-point (FP) implementation under the verified format. For these reasons, the proposed design stands as an efficient hardware engine for high-speed DSP accelerators. Future work will focus on extending the physical characterization to signed configurations and wider bit-widths to further demonstrate the scalability of the architecture.

Author Contributions: Conceptualization, G.G.-R. and R.P.-M.; Methodology, G.G.-R., R.P.-M. and A.R.-G.; Software, G.G.-R.; Validation, G.G.-R., R.P.-M., E.R.-A. and A.R.-G.; Investigation, G.G.-R., R.P.-M., E.R.-A. and A.R.-G.; Resources, R.J.-R.; Writing—original draft, G.G.-R., R.P.-M., A.R.-G. and R.J.-R.; Writing—review & editing, E.R.-A. and R.J.-R.; Supervision, R.P.-M., E.R.-A. and R.J.-R.; Funding acquisition, R.P.-M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Intel Grant "Development of specialized talent for ASIC design using INTEL's PDK-16nm, 2023" and by the COECYTJAL Project No. 10323.

Data Availability Statement: Data are contained within the article.

Acknowledgments: The authors would like to thank Intel Corporation and the technical staff from the Intel University Shuttle Program for providing technical support and access to the 16 nm technology node used in this research. The authors also acknowledge the PROFAPI program of the ITSON for its support. Finally, the authors would like to thank Juan Diego García for his valuable participation in this work.

Conflicts of Interest: Author Alberto Rodriguez-García was employed by the company Intel Corporation. Author Rodrigo Jaramillo-Ramírez was employed by the company Circuify Semiconductors. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. Vázquez-Castillo, J.; Castillo-Atoche, A.; Carrasco-Alvarez, R.; Longoria-Gandara, O.; Ortegón-Aguilar, J. FPGA-based hardware matrix inversion architecture using hybrid piecewise polynomial approximation systolic cells. *Electronics* **2020**, *9*, 182. [CrossRef]
2. Chen, Y.L.; Zhan, C.Z.; Jheng, T.J.; Wu, A.Y. Reconfigurable Adaptive Singular Value Decomposition Engine Design for High-Throughput MIMO-OFDM Systems. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2013**, *21*, 747–760. [CrossRef]
3. Omran, S.S.; Abdul-abbas, A.K. Design and implementation of 32-Bits MIPS processor to Perform QRD Based on FPGA. In *Proceedings of the 2018 International Conference on Engineering Technology and their Applications (IICETA)*; IEEE: New York, NY, USA, 2018; pp. 36–41.
4. Liu, C.; Tang, C.; Xing, Z.; Yuan, L.; Zhang, Y. Hardware Architecture Based on Parallel Tiled QRD Algorithm for Future MIMO Systems. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2017**, *25*, 1714–1724. [CrossRef]
5. Moradi Cherati, S.; Jaberipur, G.; Sousa, L. Sparse Matrix-Vector Multiplication Based on Online Arithmetic. *IEEE Access* **2024**, *12*, 87653–87664. [CrossRef]
6. Aguilera-Galicia, C.R.; Longoria-Gandara, O.; Guzmán-Ramos, O.A.; Pizano-Escalante, L.; Vázquez-Castillo, J. IEEE-754 Half-Precision Floating-Point Low-Latency Reciprocal Square Root IP-Core. In *Proceedings of the 2018 IEEE 10th Latin-American Conference on Communications (LATINCOM)*; IEEE: New York, NY, USA, 2018; pp. 1–6. [CrossRef]
7. Bharathi, M.; Mohanarangam, K.; Shirur, Y.J.M.; Choi, J.R. Accelerating DSP Applications on a 16-Bit Processor: Block RAM Integration and Distributed Arithmetic Approach. *Electronics* **2023**, *12*, 4236. [CrossRef]
8. Baungarten-Leon, E.I.; Ortega-Cisneros, S.; Jaramillo-Toral, U.; Rodriguez-Navarrete, F.J.; Pizano-Escalante, L.; Panduro, J.J.R. Vector Accelerator Unit for Caravel. *IEEE Embed. Syst. Lett.* **2024**, *16*, 73–76. [CrossRef]
9. Jaiswal, M.K.; So, H.K.H. PACoGen: A Hardware Posit Arithmetic Core Generator. *IEEE Access* **2019**, *7*, 74586–74601. [CrossRef]
10. Ercegovac, M.D.; Lang, T. *Digital Arithmetic (The Morgan Kaufmann Series in Computer Architecture and Design)*, 1st ed.; Morgan Kaufmann: Burlington, MA, USA, 2003.
11. Muller, J.M.; Muller, J.M. *Elementary Functions*; Springer: Berlin/Heidelberg, Germany, 2006.

12. Hanuman, C.; Kamala, J.; Aruna, A.R. Implementation Of Multi-Precision Floating Point Divider for High Speed Signal Processing Applications. *J. Supercomput.* **2019**, *1*, 6038–6054. [CrossRef]
13. Ebrahimi, Z.; Zaid, M.; Wijtvliet, M.; Kumar, A. RAPID: AppRoximAte Pipelined Soft MultIpliers and Dividers for High-Throughput and Energy-Efficiency. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2022**, *42*, 712–725. [CrossRef]
14. Baesler, M.; Voigt, S.; Teufel, T. FPGA Implementations of Radix-10 Digit Recurrence Fixed-Point and Floating-Point Dividers. In *2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*; IEEE: New York, NY, USA, 2011; pp. 13–19.
15. Lang, T.; Antelo, E. Radix-4 Reciprocal Square-Root and Its Combination with Division and Square Root. *IEEE Trans. Comput.* **2003**, *52*, 1100–1114.
16. Liu, M.; Fu, W.; Xia, J. Low-Latency and Minor-Error Architecture for Parallel Computing XY-like Functions with High-Precision Floating-Point Inputs. *Electronics* **2022**, *11*, 69. [CrossRef]
17. Koren, I. *Computer Arithmetic Algorithms*, 2nd ed.; CRC Press: Boca Raton, FL, USA, 2001.
18. Vera, A.; de Frutos, J.; Vega-Rodríguez, M.A. Square Root Hardware Algorithms. *Electronics* **2021**, *10*, 1988. [CrossRef]
19. Nenadic, N.M.; Mladenovic, S.B. Fast Division on Fixed-Point DSP Processors Using Newton-Raphson Method. In *EUROCON 2005—The International Conference on Computer as a Tool*; IEEE: New York, NY, USA, 2005; pp. 705–708.
20. Walczyk, C.J.; Moroz, L.V.; Samotyy, V.; Cieśliński, J.L. Optimal approximation of the 1/x function using Chebyshev polynomials and magic constants. *ACM Trans. Math. Softw.* **2024**, *51*, 2.
21. Rodríguez-García, A.; Pizano-Escalante, L.; Parra-Michel, R.; Longoria-Gandara, O.; Cortez, J. Fast Fixed-Point Divider based on Newton-Raphson Method and piecewise polynomial approximation. In *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*; IEEE: New York, NY, USA, 2013.
22. Libessart, E.; Arzel, M.; Lahuec, C.; Andriulli, F. A scaling-less Newton Raphson pipelined implementation for a fixed-point reciprocal operator. *IEEE Signal Process. Lett.* **2017**, *24*, 789–793.
23. Öztürk, E. Design and Implementation of a Low-Latency Modular Multiplication Algorithm. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2020**, *67*, 1902–1911. [CrossRef]
24. Yan, D.; Wang, W.X.; Zhang, X.W. High-performance matrix eigenvalue decomposition using the parallel Jacobi algorithm on FPGA. *Circuits Syst. Signal Process.* **2023**, *42*, 1573–1592.
25. Huang, Z.; Zhang, S.; Wang, W. An Efficient Method of Parallel Multiplication on a Single DSP Slice for Embedded FPGAs. *IEEE Access* **2019**, *7*, 100993–101008. [CrossRef]
26. Pizano-Escalante, L.; Parra-Michel, R.; Castillo, J.V.; Longoria-Gandara, O. Fast bit-accurate reciprocal square root. *Microprocess. Microsyst.* **2015**, *39*, 74–82. [CrossRef]
27. Muller, J.M. *Elementary Functions: Algorithms and Implementation*, 2nd ed.; Birkhäuser: Basel, Switzerland, 2005.
28. Synopsys, Inc. *Design Compiler User Guide, Version S-2023.03*; Synopsys, Inc.: Mountain View, CA, USA, 2023.
29. Synopsys, Inc. *IC Compiler II Implementation User Guide, Version T-2023.03*; Synopsys, Inc.: Mountain View, CA, USA, 2023.
30. Weste, N.; Harris, D. *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed.; Addison-Wesley: Boston, MA, USA, 2011.
31. Chen, S.-Y.; Wang, D.-H.; Zhang, T.-J.; Hou, C.-H. Design and Implementation of a 64/32-bit Floating-point Division, Reciprocal, Square root, and Inverse Square root Unit. In *2006 8th International Conference on Solid-State and Integrated Circuit Technology Proceedings*; IEEE: New York, NY, USA, 2006; pp. 1976–1979.
32. Bokhari, S.; Sikka, S.; Manna, R. Technology scaling trends and the future of CMOS. *Proc. IEEE* **2015**, *103*, 230–247.