Computers and Electrical Engineering 90 (2021) 106947

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

A new floating-point adder FPGA-based implementation using RN-coding of numbers

Túlio Araujo [a], Matheus B.R. Cardoso [a], Erivelton G. Nepomuceno [c], Carlos H. Llanos [d], Janier Arias-Garcia [a,b,*]

[a] Department of Electronic Engineering, Federal University of Minas Gerais, Belo Horizonte, 31270-901, MG, Brazil
[b] Graduate Program in Electrical Engineering, Federal University of Minas Gerais, Belo Horizonte, 31270-901, MG, Brazil
[c] Control and Modeling Group (GCOM), Department of Electrical Engineering, Federal University of São João del-Rei, São João del-Rei, MG, 36307-352, Brazil
[d] Department of Mechanical Engineering, University of Brasília, Brasília, D.F., 70910-900, Brazil

ARTICLE INFO

Keywords:
Round-to-Nearest representation
RN-coding
Floating-point
Adder
FPGA
Hardware

ABSTRACT

A well-known problem in the computer science area is related to numerical data representation, which directly affects adder circuits' design and a reason to have different formats: IEEE Std. 754, Half-Unit-Biased (HUB), and Round-to-Nearest (RN). RN has an advantage that rounding to nearest is equivalent to a word truncation. It avoids double rounding errors and intermediate rounding steps with an exact conversion between formats, making it applicable to general problems. However, there is a lack of research on the hardware implementation of the RN representation. In this work, we propose hardware architectures for binary and floating-point adders, analyzing for the latter its performance in terms of error and resource consumption in FPGAs. To accomplish this, we have developed a one-bit RN-based adder that allows modular designs, considering an efficient signal propagation to obtain new architectures for both binary and floating-point single-precision adders. The results open new perspectives for further applications.

1. Introduction

In computer science (CS) areas as well as in physics, chemistry, biology, among others, numerical data representation is an important aspect and a well-known problem to deal with. From the beginning, CS problems involve different challenges when representing and manipulating data for a given application. One of these challenges is related to the accuracy issue that has always been present in the computational representations because not every infinite number (continuous set) can be exactly represented in a finite set [1]. When this representation issue happens, a rounding technique is a solution to represent that number finitely, at the expense of a loss in accuracy [2]. This kind of loss is commonly called round-off error or rounding error.

Nowadays, the main format to represent floating-point numbers in CS applications is the IEEE Std. 754 (IEEE Standard for Binary Floating-Point Arithmetic) with single, double, or customized precision as the system requires. Additionally, the representation of the normal numbers, commonly known as the normalization operation, is one of the main characteristics of the IEEE-754 standard for binary numbers [3]. However, round-off errors such as the double rounding error appear in a system when a number is sequentially rounded twice. As a possible consequence, the result is not the same as rounding once to the end precision [4]. These problematic

* Corresponding author at: Department of Electronic Engineering, Federal University of Minas Gerais, Belo Horizonte, 31270-901, MG, Brazil.
E-mail addresses: tulio-@ufmg.br (T. Araujo), matheusbonavite@ufmg.br (M.B.R. Cardoso), nepomuceno@ufsj.edu.br (E.G. Nepomuceno), llanos@unb.br (C.H. Llanos), janier-arias@ufmg.br (J. Arias-Garcia).

https://doi.org/10.1016/j.compeleceng.2020.106947
Received 4 June 2020; Received in revised form 14 December 2020; Accepted 15 December 2020
Available online 29 December 2020

effects can be frequently found when the IEEE Std. 754 format is used [5]. Also, current users have to face the problem occasionally when a more/less precise representation is necessary to maintain the accuracy/silicon area specifications and when several floating-point formats are supported in a given environment. In this case, it is not easy to know in which format some operations are performed [4].

As part of the heart of a large number of CS applications, *adders* (implementing both addition and subtraction) are one of the most important arithmetic circuits used in computer architectures, including classical von Neumann based microprocessors, Digital Signal Processors (DSPs), Graphics Processing Units (GPUs), or even in data-flow based architectures such as those based on a homogeneous network of tightly coupled Data Processing Units (DPUs), called Systolic Arrays. These are often hard-wired for specific tasks, such as multiply and accumulate operations, to yield massively parallel integration, convolution, correlation, matrix multiplication or data sorting tasks. In general, adders are present in complex matrix operations, which are some of the most critical computations because they are involved in different technical applications over a wide variety of exact computation subjects around CS problems. These applications very often deal with a large amount of data, implying high computational costs and also using floating-point arithmetic operations where rounding errors appear [2].

Although there is a considerable endeavor to find a general adder circuit solution, all of them go in different directions, from solutions addressing shifting from 2D to 3D chip manufacturing to improve performance [6], to those that deal with new carry chain designs [7]. In this context, in the early years, many ways of approximating science problems (based on the representation of a real number) on computers have been introduced. Other solution approaches treated the problem dealing with particular deterministic designs that produce imprecise results exploring its error resilience with approximate computing techniques [8,9]. Also, some approaches involve, for instance, implementations of floating-point operators with different number representations, in which numerical results for a given operation will be identical to the result from an IEEE-754 compliant operator with support for round-to-nearest even [10]. Likewise, some works tried to demonstrate novel designs for floating-point adders [11–15], or proposed special Floating-point structures, generally named as fused Floating-Point Data-Path, avoiding the normalization operation. However, these approaches produced an overhead in hardware resources derived from this process because it needs to carry out after each arithmetic operation, improving performance but at the expense of a loss in accuracy, [16–18].

Despite this, the major dilemma here is to derive the best trade-off among the key design parameters such as precision, performance, area, power, and so on, when dealing with different data formats for general computer solutions to develop new algorithms to approach CS problems [19]. As this is far from being a straightforward task to be solved, adequately choosing parameters (radix, precision, exponents, etc.) to find an optimal trade-off point (or even better, making clever optimizations to design new adder circuits based on different number representation) is still an open question for numerical applications to solve general CS problems.

Given this overview, an alternative to the IEEE Std. 754 floating-point representation is the Round-to-Nearest representation of numbers, abbreviated as RN-coding. Once it is not a well-known non-standardized number system with potential advantages for CS applications, it is worth studying its scope under Adder circuit designs. Unlike the logarithmic number system, which represents numbers by their logarithms, the addition is less complex to implement [19], and the rounding to nearest is equivalent to a word truncation [20]. It avoids the double-rounding phenomenon, intrinsic to other types of representations, as well as intermediate rounding steps, potentiating its use in architecture with Fused Floating-Point Data-Path.

Lately, Half-Unit-Biased (HUB) format [21] has emerged as an alternative number representation, based on the Round-to-Nearest representation [20], showing some advantages for DSP, industrial control, or physic simulation applications. Briefly, HUB formats and canonical RN-representation present the same complexity to perform round-to-nearest and radix complement. Although RN representation numbers need at least one bit more than HUB numbers to reach the same precision, conversion from conventional formats, trivial in IEEE Std. 754 and RN cases, are not exact for HUB formats. Additionally, the authors stated that the HUB format does not offer better performance in the issue of cryptography or related applications in which exact computation is required [21]. Therefore, it is not suitable for general CS problems.

Taking these advantages of RN representation and a lack of previous works in this direction, this paper presents two new hardware architectures, one for a binary adder and the other for a floating-point adder, both based on the RN-coding of numbers. With a view to accomplish this, a one-bit RN-based adder that allows modular designs, considering an efficient signal propagation, was developed. The main goal of this work is to analyze the potential of the implementation of this number system for a new floating-point adder, using as main core the developed binary adder. This adder could be appropriate for fused data-path operations [18] such as Fused Multiply-Add (FMA) or Fused Multiply-Accumulate (FMAC), in addition to its performance in terms of precision and resource consumption against some well-known adder libraries: VFLOAT (the IEEE-754 Standard representations) and FloPoCo.

Thus, the contributions of this paper are:

- A new hardware architecture to compute the binary arithmetic addition based on the RN Representation.
- A new hardware architecture that uses the binary adder to compute the arithmetic floating-point addition based on the RN Representation.
- An analysis of the resource usage and the relative error of this architecture results for different data sets, fixed in single precision.
- A state-of-the-art comparison with well-established floating-point libraries and implementations, such as VFLOAT and FloPoCo against the proposed design implemented with the RN-coding representation in terms of precision, resource consumption on reconfigurable hardware.

Table 1
Sign alternation of the RN representation of a given number x for two significant bits.

b_i	b_{i-1}	b_{i-2}	$\delta_i = b_{i-1} - b_i$	$\delta_{i-1} = b_{i-2} - b_{i-1}$
0	0	0	0	0
0	0	1	0	1
0	1	0	1	−1
0	1	1	1	0
1	0	0	−1	0
1	0	1	−1	1
1	1	0	0	−1
1	1	1	0	0

The rest of this paper is organized as follows: Section 2 explains the basic concepts of Round-to-Nearest Representation as a brief overview, while Section 3 will address the core structure used to perform the binary addition of RN-coded numbers. Then, an explanation of the details of the proposed architecture named RN Floating-Point Adder is given in Section 4. Results are given in Section 5. Conclusions in Section 6.

2. Round-to-Nearest Representations

The RN-coding is a class of number representations introduced by [20], allowing to avoid the double-rounding errors by a simple word truncation. The double-rounding issue is a phenomenon that appears in a system that needs to deal with different levels of precision, making the result of a sequence of arithmetic operations hard to predict. When a number is sequentially rounded twice, passing for several formats supported in each environment, the result is not the same as rounding once to the end precision but the case of directed rounding [4]. This double-rounding problem occurs with some rounding methods, e.g., with the standard IEEE round-to-nearest-even.

An RN-coding number x can be achieved through the Booth recoding of its $2's$ complement representation [22], in which the value represented by:

$$x \sim b_m b_{m-1} b_{m-2} \ \cdots \ b_{l+1} b_l$$

with $b_i \in \{0, 1\}$ and $m > 1$ can be RN represented by the digit string:

$$\delta_m \delta_{m-1} \ \cdots \ \delta_{l+1} \delta_l \text{ with } \delta_i \in \{-1, 0, 1\}$$

defined by the Booth recoding for $i = l, \ \cdots \ , m$ as

$$\delta_i = b_{i-1} - b_i \ \left(\text{with } b_{l-1} = 0 \text{, following the Booth recoding}\right).$$

It is important to note that, because of the Booth recoding process, the RN representation of a given number x will always be so that any significant δ digit will be of the opposite sign of its significant adjacent digits, defining a sign alternation intrinsic to this process (see Table 1).

As defined by [22], a binary RN-coded number can have two finite representations, where one has its least significant nonzero digit equal to 1, and the other has its least significant nonzero digit equal to −1. These representations differ in the way a given number is rounded: if the least significant nonzero digit is positive, then the number was rounded up. Oppositely, if the least significant nonzero digit is negative, then the number was rounded down.

To define the "Round-to-Nearest Representation", let β be an integer such that $\beta \geq 2$. The digit sequence

$$D = d_n d_{n-1} d_{n-2} \ldots \left(\text{with } \ -\beta + 1 \leq d_i \leq \beta - 1\right) \tag{1}$$

is a RN-coding in radix β of x if

1. $x = \sum_{i=-\infty}^{n} d_i \beta^i$ (where this D is a radix β representation of x);
2. for any $j \leq n$,

$$\left| \sum_{i=-\infty}^{j-1} d_i \beta^i \right| \leq \frac{1}{2} \beta^j,$$

that is, if we truncate the digit sequence to the right at any position, the obtained sequence is always the number of the form $d_n d_{n-1} d_{n-2} d_{n-3} \ \cdots \ d_j$ that is closest to x.

Hence, truncating the RN representation of a number at any position is equivalent to round it to the nearest representable value. For every bit in the RN-coded number the value is always the nearest to the final value.

As defined for the "Round-to-Nearest Representation" in Eq. (1), the digit sequence of a number x is in error by

$$|d.d \cdots d - (x/\beta^e)| \, \beta^{n-1}$$

Fig. 1. RN Floating Point Format Representation for Single Precision. With 1 bit for the sign, 8 bits for the exponent, and 23 bits for the RN-coded Mantissa.

units in the last place (ulps) with n being the precision. An ulp of a given floating-point number x is the value represented by its least significant digit in radix β, calculated as

$$ulp(x) = \beta^{\max(e,e_{min})-n+1}$$

if $\beta^e \leq |x| \leq \beta^{e+1}$ [2], where e refers to the exponent, an integer, satisfying $e_{min} \leq e \leq e_{max}$ for $e_{min} < 0$ and $e_{max} > 0$ (which are the minimum and maximum exponents, respectively, in a given floating-point representation). If the result of a calculation is the floating-point number nearest to the correct result, it still might be in error by exact $1/2$ ulp and the relative error ranges between

$$\frac{1}{2}\beta^{-n} \leq \frac{1}{2}ulp \leq \frac{\beta}{2}\beta^{-n}$$

maintaining the same relationship with the IEEE Std. 754 format [1,3].

Besides, since there are β^{n-1} and $e_{max} - e_{min} + 1$ possible mantissas and exponents, respectively, the floating-point RN Representation can be encoded in an RN-coded number in

$$\left\lceil log_2\left(e_{max} - e_{min} + 1\right)\right\rceil + \left\lceil log_2\left(\beta^{n-1}\right)\right\rceil$$

bits, plus one bit used as a sign bit. The $n-1$ exponent of the β radix occurs because of the bit added during the Booth recoding previously described, in such a way that an RN-coded number needs one more bit in its representation. Therefore, a single-precision floating-point RN-coded number can be represented by the format described in Fig. 1.

Furthermore, the RN-coding allows for: (a) constant time, that is, the size of the representation does not influence the operation time, (b) sign inversion, that is, by inverting the sign bit as the IEEE Std. 754, and (c) rounding by truncating the representation at any point. Additionally, this approach permits previous rounding information to be passed along on arithmetic operations directly and straightforwardly, by observing the least significant nonzero digit and presuming the rounding. If this rounding occurred, it was made by truncating a partial value of the opposite sign, respecting the sign alteration at significant values achieved by the Booth recoding.

Example 1. Given the fixed point RN representation number x, where the least significant non-null bit is negative (representing the negative using an overline $\overline{d}$ on the digit),

$$x = 1.0\overline{1}01\overline{1}00101\overline{1}$$

If x is truncated at the last position, it can be seen that the least significant non-null bit of the new truncated representation is now positive:

$$\text{truncated}(x) = 1.0\overline{1}01\overline{1}0010.$$

Meaning that if a rounding occurred on the original RN number x, we can state with certainty that the number was rounded up because the only possible part of the representation that can be truncated (after the positive significant bit) must be negative, so it matches the alternating signs of an RN-coded number.

3. RN binary addition

3.1. The overall strategy

In this section, the architecture used in the binary addition of RN-coded numbers is presented in Fig. 2. In the proposed addition, numbers are encoded in a Signed Magnitude Representation, where the most significant bit represents the sign, and the following bits depicts the magnitude of the number. In this specific representation, the sign bit corresponds to the sign of the most significant bit of the magnitude, where the sign of the remaining bits follows the sign switching behavior explained in Section 2.

Because of this characteristic, before performing the RN addition operation, an architecture hereafter referred to as the *RN Signal Generator* (see Fig. 2) is utilized to calculate the sign of each bit in the number's magnitude. Given an RN represented binary number $X = r_n x_n x_{n-1} \ldots x_1 x_0$ where the r_n is the sign bit, $x_n \ldots x_0$ is the magnitude and, consequently, r_n represents the sign of x_n, the following equation can be used to calculate the sign of each magnitude bit:

$$r_i = r_{i+1} \oplus x_{i+1} \;\; \forall \; i \in \mathbb{N}, n > i \geq 0. \tag{2}$$

Following Eq. (2), it is possible to design a system that calculates the XOR operations, in which r_{i-1} would be passed to the following XOR cell to calculate r_i, acting like the carry-in bit.

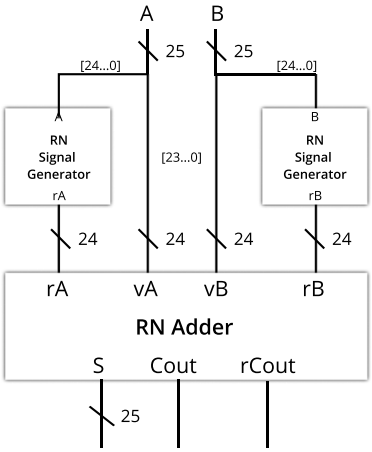

Fig. 2. Overall *RN Binary Adder* architecture design used in the addition of the binary RN-coded numbers. Two main units are presented, the *RN Signal Generator* and the *RN Adder*, as well as their inputs and outputs signals.

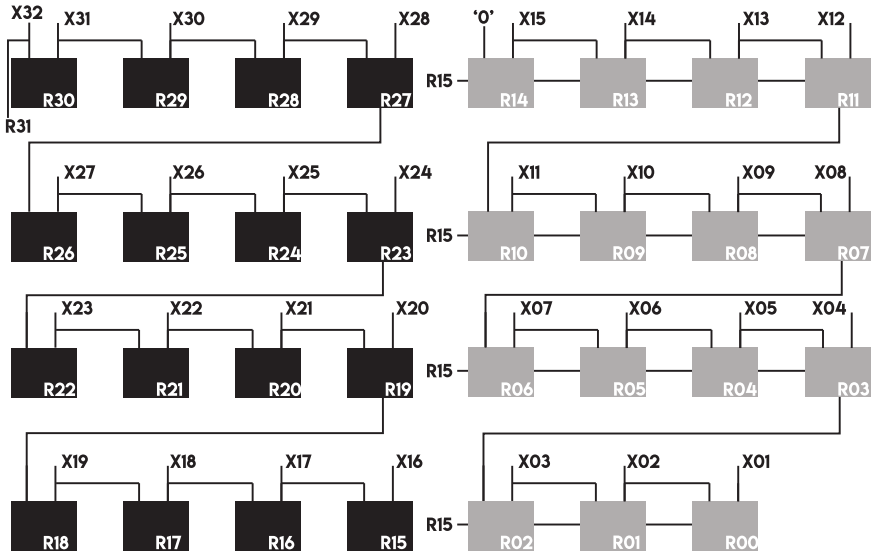

Fig. 3. Topology used to calculate the sign of each magnitude bit by XOR operations following Eq. (2). Black boxes are XOR operations between X's and gray boxes are XOR operations between X's and R15.

3.2. Tackling the signal problem

Fig. 3 shows our design approach to calculate the signs of X to reduce the well-known ripple propagation delay time. This approach takes advantage of the associative property of the XOR operation by calculating parts of each sign separately, such that the sign calculation for each magnitude bit is done in parallel computing time.

Thus, Eq. (2) is broken into two major branches, the first one having the most significant half from X, and the other containing the remaining half. These two branches work in parallel, such that the first branch result is final, while the result from the second branch is inputted in an XOR alongside the result from the least significant bit of the previous branch.

Also, to further increase this unit's performance, each major branch was divided into four smaller branches. This was done by calculating some r_i bits by expanding Eq. (2) until r_i is written only in terms of the bits in X, such that the output is achieved much faster.

3.3. Getting the RN Full Bit Adder

With the signs fully mapped to each bit in the magnitude, the binary addition is performing by the *RN Adder* unit. Before detailing this unit, the *RN Full Bit Adder* architecture, represented in Fig. 4, is explained.

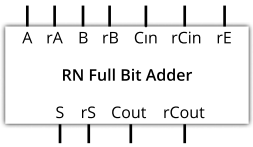

Fig. 4. *RN Full Bit Adder* architecture, used to perform the sum of two bits along a carry-in in the proposed RN representation, following the resulting Eqs. (3)–(5). As inputs the architecture receive 7 signals, named as: A, rA, B, rB, Cin, rCin and rE, producing as outputs: S, rS, Cout and rCout.

This full adder applies a combinational logic simplification of the addition of two RN-coded bits along with a carry-in to output the result S, and a carry-out, where the prefix r in inputs and outputs indicates they are a sign bit, where 0 is positive, and 1 is negative. The rE input stands for the expected result sign for this adder, being used to maintain the sign alternation intrinsic to the RN representation used in the architecture, as explained in Section 2.

A truth table was elaborated in such a way that the addition is done by the combinational logic associating A, rA, B, rB, Cin and $rCin$ to reflect the standard sum of three signaled digits in the interval $\{-1, 0, 1\}$, while respecting the expected result rE. The following equations used to implement the *RN Full Bit Adder* were achieved by simplifying the truth table of the addition:

$$S = \overline{A}\,\overline{B}\,C_{in} + \overline{A}\,B\,\overline{C_{in}} + A\,\overline{B}\,\overline{C_{in}} + A\,B\,C_{in} \tag{3}$$

$$\begin{aligned} rS = {} & \overline{A}\,\overline{B}\,\overline{C_{in}}\,\overline{rE} + \overline{A}\,\overline{B}\,C_{in}\,rE + \overline{A}\,B\,\overline{C_{in}}\,rE + \\ & A\,\overline{B}\,\overline{C_{in}}\,rE + \overline{A}\,B\,C_{in}\,\overline{rC_{in}} + A\,\overline{B}\,C_{in}\,\overline{rC_{in}} + \\ & A\,\overline{rA}\,B\,\overline{rB} + A\,B\,C_{in}\,rE + A\,\overline{rA}\,B\,\overline{rC_{in}}\,\overline{rE} + \\ & A\,B\,\overline{rB}\,\overline{C_{in}}\,\overline{rE} \end{aligned} \tag{4}$$

$$\begin{aligned} C_{out} = {} & C_{in}\,rC_{in}\,\overline{rE} + \overline{A}\,C_{in}\,\overline{rC_{in}}\,rE + A\,\overline{rA}\,B\,\overline{rB} + \\ & \overline{A}\,B\,\overline{rB}\,\overline{C_{in}}\,rE + \overline{A}\,B\,rB\,\overline{C_{in}}\,\overline{rE} + \\ & A\,\overline{rA}\,\overline{B}\,\overline{C_{in}}\,rE + A\,rA\,\overline{B}\,\overline{C_{in}}\,\overline{rE} + \\ & A\,rA\,B\,rB\,\overline{C_{in}} + \overline{A}\,B\,\overline{rB}\,C_{in}\,\overline{rC_{in}} + \\ & \overline{A}\,B\,rB\,C_{in}\,rE + A\,\overline{rA}\,\overline{B}\,C_{in}\,\overline{rC_{in}} + \\ & A\,rA\,\overline{B}\,C_{in}\,rE + A\,rA\,B\,rB\,\overline{rE} \end{aligned} \tag{5}$$

$$rCout = A\,rA + B\,rB + \overline{A}\,\overline{B}\,\overline{rE}. \tag{6}$$

3.4. The proposal for the RN Adder

Given that the *RN Full Bit Adder* unit takes advantage of the alternation of signs, it is important to preserve this behavior. This is done by taking advantage of the signaled digit used in the RN Representation to normalize the addition result. In this way, the alternation of signs is preserved when the expected sign is different from the resulting one. This is done using the same approach for the Booth recoding used to convert a 2's complement number to RN. Here, the equation $x = 2x - x$ can be utilized in part of the binary string, and for instance,

$$0101\overline{1} = 8 + 2 - 1 = 9$$

has the same value of

$$1\overline{1}01\overline{1} = 16 - 8 + 2 - 1 = 9$$

but the second representation preserves the sign alternation. In the proposed architecture, this is done by setting the carry-out bit to 1 with the negation of rE as its sign and setting $rS = rE$. This is the same procedure used to derive, from the truth table, Eqs. (3), (4), (5), and (6).

With the core of the addition explained, the *RN Adder* unit (see Fig. 5) previously mentioned is divided into blocks referred as *RN Adder Sets* (depicted in Fig. 6) such that n (the length of the numbers being added) must be divisible by 3, which is the number of bits being added in each set. This division, which can be parameterized for specific needs, was selected because it poses a good balance between the parallel addition inside of the set and the carry bit treatment between them.

The *RN Adder Sets* are connected between each other using the carry-out signal (*Cout*) and negated result sign (*rS*) from a block, which are passed respectively to the carry-in signal (*Cin*) and expected result sign (*rE*) of the next one. The result sign is negated for the next expected sign to guarantee the sign alternation previously mentioned.

Every set (detailed in Fig. 6) is composed of four *RN Adder Modules* (as presented in Fig. 7), each being responsible for the addition of three bits of A and B for one of the four possible combinations of *Cin* and *rE*. The correct result is chosen based on these inputs. This is done so that the performance costly sum in the *RN Adder* can be performed in parallel across all sets independently of previous results, that are used only to select the correct one in a multiplexer in each set.

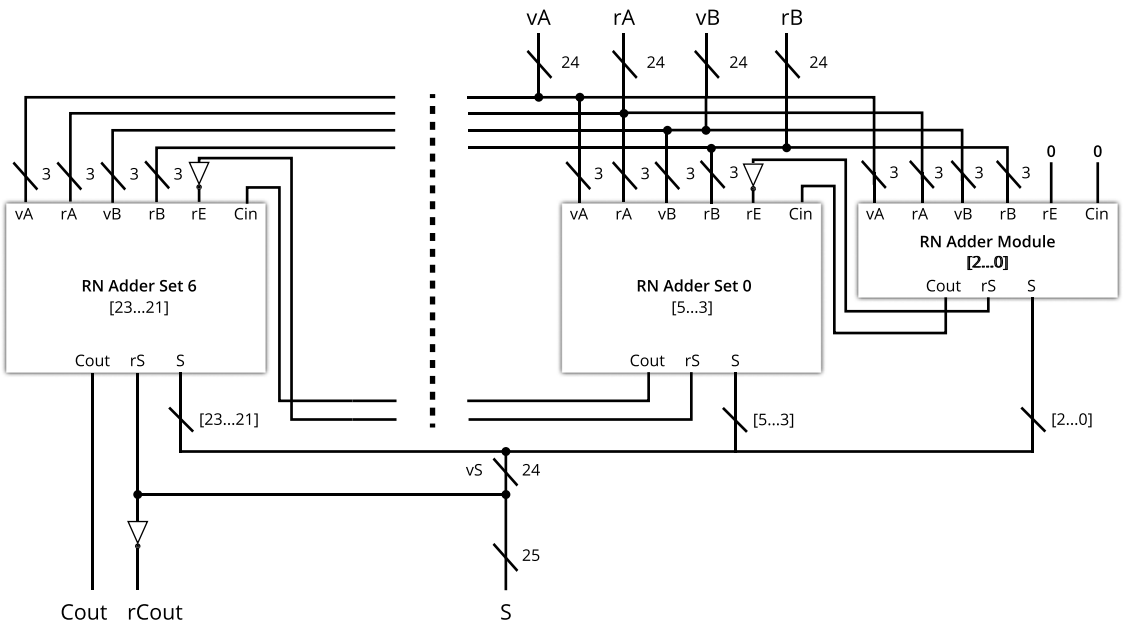

Fig. 5. *RN Adder* unit, responsible for the sum of RN-coded numbers vA and vB paired with their sign bits rA and rB, based on the *RN Adder Sets*. Each set is composed of four *RN Adder Modules*, each being responsible for the addition of three bits of A and B for one of the four possible combinations of Cin and rE, where the correct result is chosen based on these inputs.

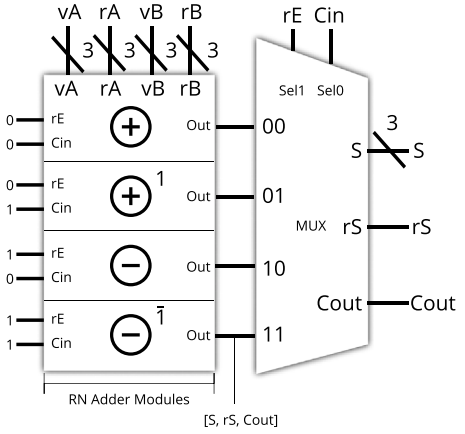

Fig. 6. *RN Adder Set* architecture, used to perform the parallel sum of three RN-coded bits for all possible combinations of the expected result and carry-in, in which the correct answer is then chosen using the outputs from a previous block.

Because the least significant bit in the addition has no expected result sign, the three least significant bits are added in the *RN Adder* using a single *RN Adder Module* with a chosen positive expected result with no carry-in. A negative expected result without carry-in could also be applied with no difference in the unit's result. Also, because of the RN representation sign alternation, each set and module in Fig. 6 will pass only the carry-in value to the next unit, and not its sign, since the expected result and carry-in must always have the same sign.

The *RN Binary Adder* architecture then outputs the concatenation of the result sign and the results of each set (and the first module) in the RN-coded value *S*, as well as the carry-out value and sign bits *Cout* and *rCout*.

Observing the *RN Binary Adder* unit detailed in Figs. 2, 5, 6, 7, a graphical scheme of the unit's execution time was developed for an arbitrary 25 bits input (a single-precision mantissa plus the hidden-bit and a sign bit), as presented in Fig. 8. By analyzing Fig. 8, it can be seen that the unit has 13 sequential steps. Each group of individual nodes has different meanings depending on the stage analyzed. Also, each node can be defined by XOR gates for the *RN Signal Generator* stage, or by *RN Full Bit Adders* for the *RN Addition* stage, or by multiplexers for the *Set Result Selection* stage in the *RN Adder Sets*.

Considering a fully connected ripple carry architecture, for both the signal generators and the adders, and a given input of 25 bits, a total of 48 sequential steps would be necessary to calculate the binary addition of two RN-coded numbers. Given this, the

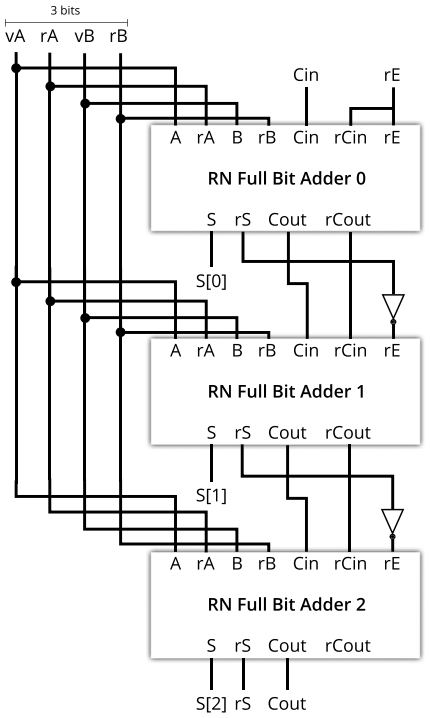

Fig. 7. *RN Adder Module*, composed of three *RN Full Bit Adders* sequentially connected to add three RN-coded bits.

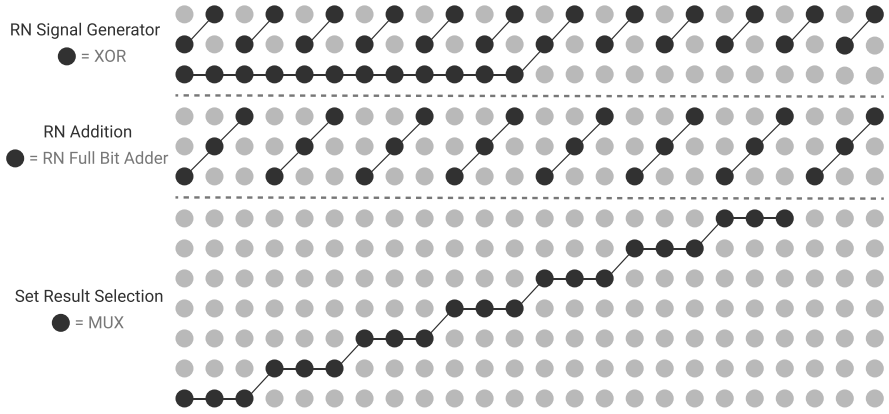

Fig. 8. Execution time graphical scheme for the *RN Binary Adder* unit, where each row is executed in parallel time, while the connections between nodes represent the passing of results from previous to the next executions step.

proposed *RN Binary Adder* architecture uses parallel execution to reduce the number of execution steps by 72.927% when compared to this hypothetical ripple carry architecture.

4. RN floating-point Adder pipelined architecture

Considering the particularities of the RN representation presented in Section 2 and the *RN Binary Adder* unit detailed in Section 3, an architecture capable of performing floating-point addition is proposed, with characteristics of a *data-flow architecture* (without FSM-based control), where a signal exchange synchronizes the flow of the pipeline stage information.

Taking advantage of the sign switching at significant values intrinsic to the RN-coding, it is possible to represent each signaled bit of a given value by merely appending a sign bit similar to the one used in the IEEE Std. 754, which corresponds to the sign of the hidden-bit appended to the mantissa when evaluating the floating-point number. Alongside the sign bit, the floating-point RN representation proposed is structured by a commonly biased exponent and an RN-coded mantissa, in which the most significant bit of the mantissa has the opposite sign of the hidden-bit.

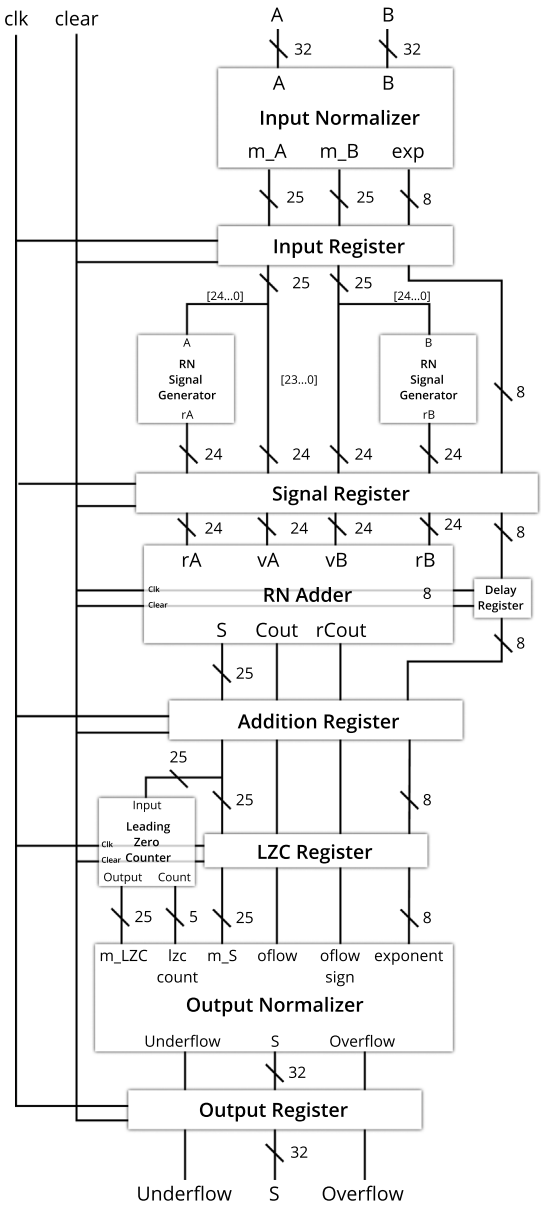

Fig. 9. *RN Floating-Point Adder* architecture design. A six-stage pipelined architecture based on the *RN Adder* was developed to execute the floating-point addition of two single-precision floating-point RN represented numbers.

Based on this, a six-stage pipelined architecture was developed to execute the floating-point addition of two single-precision floating-point RN represented numbers. The diagram of that architecture, hereafter referred to as the *RN Floating-Point Adder*, is presented in Fig. 9.

As observed in Fig. 9, the first step taken in the *RN Floating-Point Adder* architecture is to normalize the inputs using the *Input Normalizer* unit. This unit is responsible for equalizing the exponents of inputs A and B, where the mantissa of the lower input is right-shifted after the hidden-bit is appended to it, to have the same exponent of the greater input. During this process, the characteristic rounding by truncation of the RN representation is utilized to round the shifted mantissa, saving resources and potential execution time, as well as ensuring this rounding is done at constant time. The outputs of this component are then passed to the Input Register, defining the first pipeline stage of the architecture.

Following the input normalization, the extended normalized mantissas are inputted in the *RN Signal Generators* presented in Section 3, which generate the signs for each magnitude bit of the mantissa. It is output, along with the magnitude of A and B's mantissas and the exponent, is passed to the *Signal Register*, marking the second pipeline stage of the *RN Floating-Point Adder*.

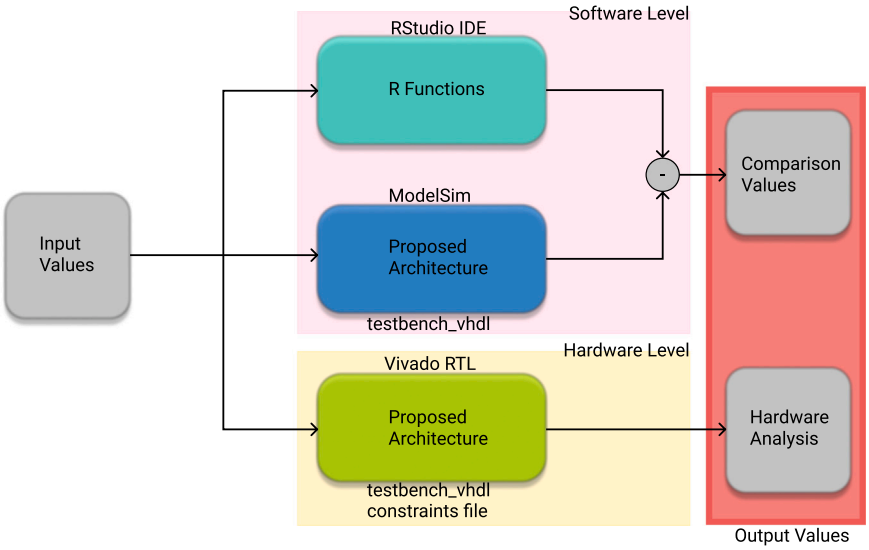

Fig. 10. Simulation and Synthesis flow details used for Software and Hardware design analysis. For Software, RStudio IDE was used as a statistical estimator. For Hardware analysis, the Vivado tool was used for the implementation process of the proposed architecture, as well as for a logic resource estimation.

After this, the mantissas, defined by the pairs (vA, rA) and (vB, rB), are passed to an altered *RN Adder* unit that uses one clock cycle to perform the binary addition. Because of this clock cycle, the exponent is passed through a register to synchronize its time with the rest of the signals, establishing the third pipeline stage of the architecture.

Subsequently, the outputs of the *RN Adder* unit alongside the output exponent from the *Delay Register* are then passed to the *Addition Register*, outlining the fourth pipeline stage of the *RN Floating-Point Adder*.

Thereafter, the result of the addition is passed through the *Leading Zero Counter* (LZC) unit. The LZC takes one clock cycle and is responsible for left-shifting the result for the number of leading zeros, fitting the hidden-bit in the Most Significant Bit (MSB) position, and outputting the shifted value and the shift count. In the same way that was done with the *RN Adder* unit, because of the clock cycle used by the LZC, the previous outputs from the addition are delayed using the LZC register, which marks the fifth pipeline stage.

Finally, the results from the LZC and the addition are passed to the *Output Normalizer*. This unit is used (a) to remove the hidden-bit from the resulting mantissa, (b) to normalize the exponent with the LZC shift count or the addition carry-out, (c) to concatenate the sign, exponent, and mantissa in a single output *S* and (d) to detect representation overflow and underflow, which then pass its outputs to the *Output Register*, defining the sixth and last stage of the *RN Floating-Point Adder*.

It is important to notice that, because of the characteristic rounding on the truncation of the RN representation, no rounding unit is needed in any of the steps described above, resulting in architecture free of extra rounding steps as normally presented in other libraries [23,24].

5. FPGA implementation results

The implementation of the RN *Floating-Point Adder* architecture is presented here using the Xilinx Artix 7 series $XC7A100T-1CSG324C$. The core was developed in VHDL and synthesized using Xilinx's Vivado v2017.4. After implementing the designed *RN Floating-Point Adder* architecture on FPGA, a series of tests were done to analyze it for two main parameters: precision and resource consumption.

The error analysis of the RN *Floating-Point Adder* architecture must be carefully considered showing the differences of this particular implementation and demonstrating if this basic arithmetic architecture introduces more rounding error than necessary.

Hence, along with the RStudio IDE, the R environment was used as a statistical estimator and as a graphic plotting, against double floating-point representation values. Also, numerical software results, from R and ModelSim environments, were compared to establish a relative error to measure the proposed architecture's rounding error. All selected test cases try to expose the behavior of the RN representation near boundary quantities. The general flow used for design analysis is presented in Fig. 10, in which one can observe the use of test benches for both parts of the process, the software analysis, and the hardware implementation.

Also, configuration scripts were implemented to generate all the data values for the *RN Floating-Point Adder* architecture, as presented in Fig. 10, as well as to automatically deal with the mantissa conversion process from 2's complement representation to RN representation and conversely, as shown in algorithms 1, 2 and in Fig. 11.

Therefore, five different tests were performed addressing different cases, as follows:

Algorithm 1 Conversion process from 2's complement to RN representation.

```
procedure
    b ← float2bin(single(f))
    signal ← b[1]
    exponent ← b[2 : 9]
    mantissa ← b[10 : 32]
    #Concatenate the hidden bit and
    #0 so the mantissa
    #will be treated as a positive RN number
    mantissa ← '01' + mantissa
    rnMantissa = ''
    for i in 1 : length(mantissa) − 1 do
        if mantissa[i] == mantissa[i + 1] then
            rnMantissa += '0' #1 − 1 or 0 − 0
        else
            rnMantissa += '1' #1 − 0 or 0 − 1
        end if
    end for
    rnMantissa += mantissa[length(mantissa)]
    #Remove hidden bit
    rnMantissa = rnMantissa[2 : 24]
    #Increment the exponent, as
    #the RN conversion
    #add a bit to the representation
    exponent = bin2dec(exponent)
    exponent += 1
    exponent = dec2bin(exponent, 8)
    #Concatenate the result
    rn = signal + exponent + rnMantissa
end procedure
```

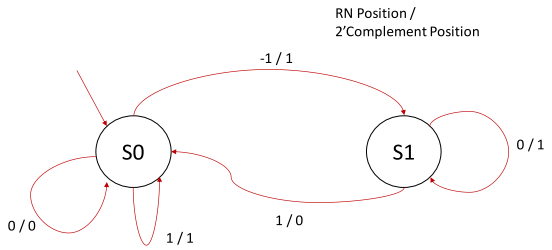

Fig. 11. Conversion Process from 2's complement to RN representation.
Source: Adapted from [20].

Table 2
Relative error linked to the addition process of the proposed RN Floating-point Adder, the FloPoCo FPAdder, and the VFLOAT Adder, showing a competitive error of the proposed approach. Values presented in scientific notation. The asterisk (*) means some inputs that have not been correctly added by the *VFLOAT* architecture.

	RN Floating-Point Adder		FloPoCo FPAdder [24]		VFLOAT Adder [25]	
	Average error	Maximum error	Average error	Maximum error	Average error	Maximum error
SOM	5.16^{-08}	2.04^{-07}	2.73^{-08}	9.93^{-08}	3.41^{-08}	1.38^{-07}
SDOM	6.53^{-08}	2.76^{-07}	2.90^{-08}	1.08^{-07}	4.68^{-08}	1.59^{-07}
GDOM	5.27^{-08}	1.72^{-07}	2.11^{-08}	5.81^{-08}	3.69^{-06*}	3.71^{-03*}
SN	8.88^{-08}	5.06^{-07}	3.65^{-08}	2.13^{-07}	3.97^{-08}	2.13^{-07}
BN	8.66^{-08}	6.94^{-07}	3.69^{-08}	2.58^{-07}	4.21^{-08}	2.58^{-07}

1. Sum of numbers of the same order of magnitude (SOM), ranging from 1.99^{-01} to 8.97^{-01};
2. Sum of numbers with a small difference in the order of magnitude (SDOM), ranging from 5.02^{-05} to 4.00^{-01};
3. Sum of numbers with a great difference in the order of magnitude (GDOM), ranging from 2.00^{-08} to 9.99^{07};

Algorithm 2 Conversion process from RN representation to 2's complement.

```
procedure //
    signal ← rn[1]
    exponent ← rn[2 : 9]
    rnMantissa ← rn[10 : 32]
    #Cast to integer
    signal = int(signal)
    #Concatenate thehidden bit
    rnMantissa = '1' + mantissa
    mantissa = 0
    s = 1
    for i in 1 : length(rnMantissa) do
        if rnMantissa[i] == '1' then
            mantissa += s * 2^(1 − i)
            s *= −1
        end if
    end for
    #Remove hidden bit
    rnMantissa = rnMantissa[2 : 24]
    #Convert Exponent to Decimal and
    #remove bias
    exponent = bin2dec(exponent)
    exponent −= 127
    f = single(signal * mantissa * 2^exponent)
end procedure
```

4. Sum of small numbers, close to the lower limit of the representation (SN), ranging from 2.00^{-27} to 9.99^{-27};
5. Sum of big numbers, close to the upper limit of the representation (BN), ranging from 2.00^{30} to 9.98^{+30}.

In the overall context of this work, values larger than $\beta \times \beta^{e_{max}}$ or smaller than $1.0 \times \beta^{e_{min}}$ are not considered in the analysis. As a matter of example for what is considered the relative error, when approximating 5.17165 by 5.17×10^0 the relative error involved is $0.00165/5.17165 \cong 0.0003$.

In this context, for each of these tests, 10000 single-precision samples following the IEEE Std. 754 format were used as input to the RN Floating-Point Adder, to the *FloPoCo FPAdder* [24] and to the *VFLOAT Adder* [25]. Results from these architectures were compared to the standard double-precision IEEE Std. 754 floating-point addition result, performed in software, to estimate and compare the relative error linked to the addition process of each design.

Special attention was taken so that any real number used in the tests is inside the range of representation, in order to avoid overflow and underflow errors. However, the same attention was not taken for real numbers that might not be exactly representable as a floating-point number. For instance, when the result of a decimal number converted to a binary has an infinite repeating representation, in which these cases were included in the results.

In the GDOM tests, some inputs have not been correctly added in the *VFLOAT Adder* architecture, whose results negatively influenced the error estimates for this architecture, as shown in the values marked with an asterisk (*) in Table 2.

Analyzing the results in Table 2, it can be noted that, even though the three architectures present consistent errors (except the GDOM test case for the *VFLOAT Adder*) in the addition of single-precision numbers, the FloPoCo library has a slight advantage over VFLOAT. Besides, both of them have, in most cases, half of the average relative error of the *RN Floating-point Adder* architecture. This is justified mainly by the loss of one bit of precision during the conversion of the mantissa between two's complement and the RN-coding using the Booth recoding.

Together with the resulting error estimates, the probability density of the relative error for the *RN Floating-Point Adder* architecture was calculated for the general cases. The results, presented in Fig. 12, demonstrate that the vast majority of cases contemplated in this analysis provide an associated error with values around 1E-09 to 5E-07, averaging to 6.90E-08.

Concerning the FPGA Artix, the synthesis result, presented in Table 3, has shown that the proposed architecture used about 597 (0.94%) with 6-input LUT and 439 (0.35%) FF. No internal memory, no internal LUTRAM, and no internal DSP were used in the synthesis design. Comparing with the *FloPoCo FPAdder*, it used about 376 (0.59%) LUT, 411 (0.32%) FF, as well as 10 (0.05%) LUTRAM. The *RN Floating-Point Adder architecture* requires more LUTs with a ratio of 1.58 against the *Flopoco Adder*, but *Flopoco Adder* uses LUTRAM and only implements the sum operation information as an output. While the *RN Floating-Point Adder* architecture delivers, apart from the sum, signs of underflow and overflow. Also, the total power ratio is 0.948, showing that the proposed architecture consumes less power than the *Flopoco Adder*. The VFLOAT adder used fewer resources than both architectures discussed above (see Table 3), although VFLOAT showed several errors in their precision tests. Furthermore, the system was configured to follow the strategies *Flow_PerfOptimized_high* with retiming and *Performance_Explore*, in the constraints file (see Fig. 10), for synthesis

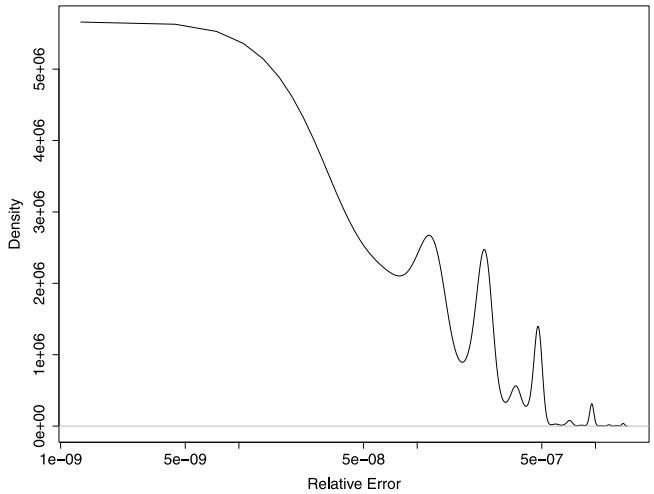

Fig. 12. Probability density of the relative error for the *RN Floating-Point Adder* architecture.

Table 3
Resource usage after synthesis of the floating-point architectures configured for single-precision representation. Three Adders architectures are compared: *RN FP Adder*, *FloPoCo FPAdder*, and *VFLOAT Adder*. The comparisons are in terms of LUT, LUTRAM, FF, and Power Consumption, showing the potential of the proposed architecture for hardware implementations.

	RN FP Adder	FloPoCo FPAdder [24]	VFLOAT Adder [25]
LUT	597	376	377
LUTRAM	–	10	2
FF	439	411	367
Total power (mW)	185	195	160

and implementation, respectively. Thereby, the RN *Floating-Point Adder* with 6 pipeline stages can achieve up to 225.2 MHz, and the *FloPoCo FPAdder* achieves a frequency of up to 243.9 MHz with 9 pipeline stages, and the *VFLOAT Adder* a frequency of up to 225.2 MHz with 5 pipeline stages.

6. Conclusion

The proposed floating-point adder architecture explored the round-to-the-nearest process by simple truncation of the values in the RN coding of numbers. This was taken as an advantage for its hardware designs and for avoiding extra round-off steps after the normalization process. It is worth noting that our architecture sets its parametrization to a mantissa in multiples of 3, configured to reduce the number of logical levels. In this way, a robust unit was created to select the best relationship of parameters (performance, area, or precision) according to each application's needs.

Although the results show a disadvantage related to the precision of the calculations, when compared with FloPoCo FPAdder, the RN representation provides one digit to load the rounding information. This implies the loss of 1/2 ulp of resolution. It is also worth mentioning that FloPoCo internally uses a 34-bit representation. However, we can highlight that RN representation avoids double rounding propagation, which potentially allows implementations with less hardware and joint arithmetic operations without intermediate denormalization/normalization steps.

Nonetheless, we must point out that even with the relative cost, the convenience of using the proposed adder lies in establishing a native RN architecture. This possibility guarantees the RN number system's properties when dealing with applications where avoiding the double rounding error can significantly increase precision. This fact opens perspectives in cases in which exact computation is required, in contrast to other approaches found in existing literature.

CRediT authorship contribution statement

Túlio Araujo: Investigation, Writing - original draft, Writing - review & editing, Software, Hardware, Validation. **Matheus B.R. Cardoso:** Investigation, Writing - original draft, Writing - review & editing, Software, Validation. **Erivelton G. Nepomuceno:** Writing - review & editing, Methodology. **Carlos H. Llanos:** Writing - review & editing, Methodology. **Janier Arias-Garcia:** Writing - original draft, Writing - review & editing, Project administration, Supervision, Methodology, Hardware, Validation, Funding acquisition.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.compeleceng.2020.106947.

Acknowledgments

The authors would like to thank *Pró-Reitoria de Pesquisa (PRPq)* of the *Universidade Federal de Minas Gerais* (UFMG), Brazil, FAPEMIG, Brazil, CNPq, Brazil, and CAPES, Brazil. The authors would also like to thank the Xilinx University Program (XUP) for the software licenses provided.

References

[1] Goldberg D. What every computer scientist should know about floating-point arithmetic. ACM Comput Surv 1991;23(1):5–48.
[2] Muller J-M, Brisebarre N, de Dinechin F, Jeannerod C-P, Lefèvre V, Melquiond G, et al. Handbook of floating-point arithmetic. Birkhäuser Boston; 2010, p. 572.
[3] Institute of Electrical and Electronics Engineers (IEEE). IEEE Standard for Floating-Point Arithmetic. 2019, IEEE Std 754-2019 (Revision of IEEE 754-2008).
[4] Martin-Dorel É, Melquiond G, Muller J-M. Some issues related to double roundings. Tech. rep., INRIA; 2011, p. 42, URL: http://hal-ens-lyon.archives-ouvertes.fr/ensl-00644408.
[5] Panhaleux A. Contributions to floating-point arithmetic : Coding and correct rounding of algebraic functions [Ph.D. thesis], Ecole normale supérieure de lyon - ENS LYON; 2012.
[6] DeBenedictis E. It's time to redefine Moore's law again. Computer 2017.
[7] Efstathiou C, Owda Z, Tsiatouhas Y. New high-speed multioutput carry look-ahead adders. IEEE Transactions on Circuits and Systems II: Express Briefs 2013.
[8] Esposito D, De Caro D, Napoli E, Petra N, Strollo AGM. Variable latency speculative han-carlson adder. IEEE Trans Circuits Syst I Regul Pap 2015.
[9] Liu W, Chen L, Wang C, O'Neill M, Lombardi F. Design and analysis of inexact floating-point adders. IEEE Trans Comput 2016;65(1):308–14.
[10] Ehliar A. Area efficient floating-point adder and multiplier with IEEE-754 compatible semantics. In: 2014 International conference on field-programmable technology (FPT); 2014. p. 131–8.
[11] Jaiswal M, Cheung RCC, Balakrishnan M, Paul K. Unified architecture for double/two-parallel single precision floating point adder. IEEE Tran Circuits Syst II: Express Briefs 2014;61(7):521–5.
[12] Jaiswal M, Varma BSC, So HKH. Architecture for dual-mode quadruple precision floating point adder. In: 2015 IEEE computer society annual symposium on VLSI; 2015. p. p. 249–54.
[13] Drusya P, Jacob V. Area efficient fused floating point three term adder. In: 2016 International conference on electrical, electronics, and optimization techniques (ICEEOT); 2016. p. 1621–5.
[14] Narule O, Palsodkar P. Implementation of three operand floating point adder. In 2016 international conference on communication and signal processing (ICCSP); 2016. p. 1037–40.
[15] Zhang H, Chen D, Ko SB. High performance and energy efficient single-precision and double-precision merged floating-point adder on FPGA. IET Comput Digit Tech 2018;12(1):20–9.
[16] Langhammer M. Floating point datapath synthesis for FPGAs. In: 2008 International conference on field programmable logic and applications; 2008.
[17] Synopsys. DWFC flexible floating point overview. 2016, URL: https://www.synopsys.com/dw/ipdir.php?ds=dw_foundation_cores0. [Online; Accessed 28 May 2020].
[18] Gonzalez-Navarro S, Hormigo J. Normalizing or not normalizing? An open question for floating-point arithmetic in embedded systems. In: 2017 IEEE 24th symposium on computer arithmetic (ARITH); 2017.
[19] Constantinides G, Kinsman A, Nicolici N. Numerical data representations for FPGA-based scientific computing. IEEE Des Test Comput 2011;28(4):8–17.
[20] Kornerup P, Muller J, Panhaleux A. Performing arithmetic operations on round-to-nearest representations. IEEE Trans Comput 2011;60(2):282–91.
[21] Hormigo J, Villalba J. New formats for computing with real-numbers under round-to-nearest. IEEE Trans Comput 2016;65(7):2158–68.
[22] Kornerup P, Muller J, Panhaleux A. Floating-point arithmetic on round-to-nearest representations. 2012, CoRR abs/1201.3914.
[23] Belanović P. LM. A library of parameterized floating-point modules and their use. In: Proceedings of the international conference on field programmable logic and applications (FPL); 2002. p. 657–66.
[24] de Dinechin F, Pasca B. Designing custom arithmetic data paths with FloPoCo. IEEE Des Test Comput 2011;28(4):18–27.
[25] Wang X, Leeser M. VFloat: A variable precision fixed- and floating-point library for reconfigurable hardware. ACM Trans Reconfig Technol Syst (TRETS) 2010;3(3):16:1–34.

Túlio Araújo de Oliveira is a Control and Automation Engineer who graduated from the *Universidade Federal de Minas Gerais* (UFMG). His interests include Software Development and Architecture, Cloud Computing, Machine Learning, Arithmetic Architecture Designs, and Embedded Systems Design. He won a certificate of honor due to his academic results on his graduation from UFMG.

Matheus B.R. Cardoso is a Control and Automation Engineer student, currently pursuing his Bachelor's Degree at the *Universidade Federal de Minas Gerais* (UFMG). His interests include Software Development, Machine Learning, Data Science, Embedded Systems, and Chaos Applications. He spent a semester as an exchange student at the Southeast Missouri State University (SEMO), awarding a position on the Dean's List (Spring 2020).

Erivelton G. Nepomuceno is IEEE Senior Member and works at Department of Electrical Engineering — UFSJ. He was postdoctoral (Imperial College) and researcher visitor (Saint Petersburg Electrotechnical University). He is associate editor for: IEEE Trans Circuits Syst II; J. Control Automation Elec. Syst; IEEE Latin Am Trans; Mathematical Problems Eng. Research topics: chaos, complex systems, computer arithmetic, interval arithmetic, system identification.

Carlos Humberto Llanos is a professor in the Mechanical Department, University of Brasilia (UnB), Brazil. His research interests include Reconfigurable Computing, Embedded Systems Design, Machine Learning, Optimization, and Automation/Control Design in Embedded System Platforms. He received his Ph.D. degree in Electrical Engineering from *Escola Politécnica da Universidade de São Paulo* (EPUSP), Brazil, 1998. He is a Member of the IEEE.

Janier Arias Garcia is a professor in the Department of Electronic Engineering at the *Universidade Federal de Minas Gerais* (UFMG). He has experience mainly in the development of architectures and custom hardware implementations, involving theory and practice of using reconfigurable hardware systems to describe digital circuits, ranging from Numerical Algorithms, Machine Learning, Chaotic Systems, and Optimization, for Embedded Systems Applications.