

Received January 8, 2021, accepted January 19, 2021, date of publication January 29, 2021, date of current version February 9, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3055735

Review of Basic Classes of Dividers Based on Division Algorithm

UDAYAN S. PATANKAR¹, (Member, IEEE), AND ANTS KOEL, (Member, IEEE)

Thomas Johann Seebeck Department of Electronics, Tallinn University of Technology, 19086 Tallinn, Estonia

Corresponding author: Udayan S. Patankar (udayan.patankar45@gmail.com)

This work was supported in part by the Estonian Research Council Institutional Research Projects under Grant IUT19-11, Grant PUT1435, and Grant PRG780, and in part by the European Union's Horizon 2020 Research and Innovation Program under Grant 668995.

ABSTRACT The electronics world is very well described in two distinct but dependent interdisciplinary areas, namely hardware and software. Arithmetic operations are very vital building blocks of an electronic system. An algorithm is a systematic arrangement that helps develop a sophisticated electronic system, including hardware and software aspects. Addition, subtraction, multiplication, and division are critical elements of arithmetic implementation in the electronic system, but fewer efforts have been made to implement division than other arithmetic operations, even though the number of transistors on a chip is increasing beyond the Moore's law prediction. It is quite complicated to implement arithmetical operations; here, a sophisticated algorithm is essential to successful implementation. Technological upgrades are leading to a new paradigm of applications, where the performance of a division circuit or block is a vital and critical feature of a successful system. The lexicon of algorithms used in the implementation of the division operation in electronics systems is discussed in detail in the present article, which indicates the mathematical formulation, criticality, conversion pattern, hardware requirements, and logic used for conversion. The current report describes the broad classification of dividers into basic classes named digit recurrence, high radix, functional iteration, estimation, a look-up table, and variable latency. It also illustrates that, in practical implementation, many algorithms have been developed that combine one or many classes and are implemented with different hardware architectures. The study indicated the possibility of improving the presently available algorithms or creating a new algorithm to enhance practical implementation.

INDEX TERMS Divider, SRT, restoring, non-restoring, digit recurrence, radix-n, FPGA, functional iteration, look-up table, variable latency.

I. INTRODUCTION

Mathematics is not just a word but has also had a colossal status in the life of human beings from its very beginnings. Sometimes it is not only an indicative word but also acts as a science of numbers and their relations or, eventually, both. The theoretical study of mathematics is specially named Theoretical Mathematics, whereas another side of it, termed Applied Mathematics, is useful in different computing aspects of daily life [1], [2]. It is no exaggeration to say that mathematics is everything and that everything is mathematics. From the very early stages of the human race, mathematics has been in force. From the beginning of our evolution, mathematics was involved in counting, time, and space; later, when humans started to understand more aspects

of life, it stimulated the study of various fields like astronomy, architecture, ratio proportions, navigation, etc., to fulfill their requirements. This gave a more significant aspect to the requirement for applied mathematics in human life; until the industrial revolution, mathematics was extensively used in chemistry, physics, architecture, metallurgy, and financial sectors. The initial phases of industrialization were reliant on the new ways of theoretical mathematics and physics in the field of industry to develop mass production techniques that could provide a better solution to economic difficulties in producing various items or products. These efforts from applied physics and mathematics gave birth to new possibilities, leading to the newborn field of electronics and integrated circuits, which has proved very valuable and innovative for existing applications like communications, transport, and calculations. In the beginning, communication was fully analog. With technological evolution, it changed to digital, but

The associate editor coordinating the review of this manuscript and approving it for publication was Gian Domenico Licciardo¹.

whether analog or digital, the concept of modern communication at that time also showed a significant dependence on mathematics. In current times, the importance of digital communication and computation has reached a different level. Which in turn allows the evolution of new fields of work and study in the data protection area, statistical data analysis, computational processing, signal processing, artificial intelligence, image processing, complex systems on chips, central processing unit, graphics processing unit, biomedical equipment, fuzzy control, space engineering, etc. [3]–[11], [52]–[56]. Fig. 1 illustrates the different trends of application. However, addition, subtraction, multiplication, and division remain vital building blocks in implementing modern theories of theoretical and applied mathematics [4], [52], [58], [59] and represent the mathematical operations’ essential properties as illustrated in Fig. 2.

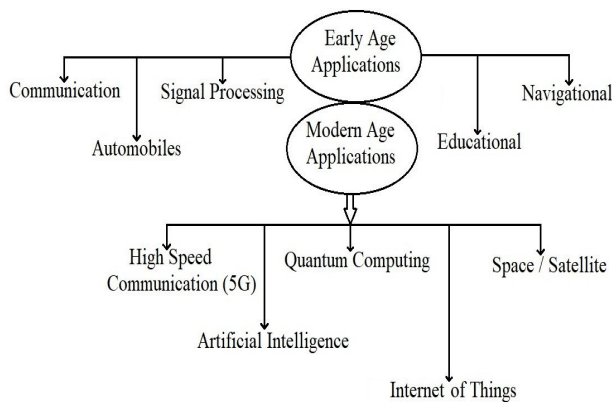


FIGURE 1. The trends of application.

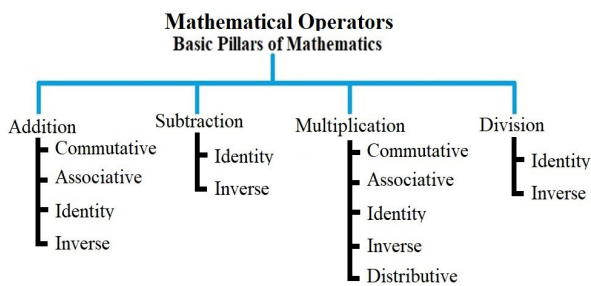


FIGURE 2. The properties of various mathematical operators.

Due to the lack of communication and transport means, there was no such typical period of evolution of mathematics around the world. In the modern era, mathematicians have researched old concepts and developed new concepts to meet today’s computational and technological enhancement requirements. Although new concepts, operations, logic, and relations have been developed in mathematics, addition, subtraction, multiplication, and division are still the strong foundation of applied mathematics [4], [52]. The addition is a simple terminology that indicates the action of collecting or grouping in general. The commutative and associative aspects

of the addition operation have made it easy to perform its electronic application from the beginning of the new electronics era [12], [13]. Subtraction is a second but very important operation. It is defined as the act of reduction. Multiplication is one of the basic operations but a derived function in mathematics. It also combines multiple quantities into a single amount like addition. Multiplication is also known as successive addition. The division operation is also a derived operation like multiplication; instead of successive addition, it involves successive subtractions along with some controlling conditions. The result of the successive subtractions must be tested under several controlling conditions before its finalization. It has a high dependency on the order of two quantities connected by the division operator. Unlike multiplication and addition, the division operation does not possess commutative and associative properties, making it critical and challenging to implement in an electronic way [3]–[11].

Fig. 3 (a), (b), and (c) show the fundamental ways of performing division operations using (a) the successive subtraction method, which is also called a long division algorithm or paper and pencil algorithm, and (b), (c) the look-up table method. The successive subtraction method looks very easy and possesses a simple quotient conversion logic; when it comes to implementing electronically for critical systems, it is not suitable to use simple recursive logic for conversion. Thus, many methods or algorithms have been researched over a period to implement an efficient divider for an efficient system. There exist various algorithms to perform division in multiple ways. Still, broadly, depending on the logic of the quotient conversion, they can be summarised into multiple divider classes, which are discussed and compared in detail in the next sections of this article. Selection of the appropriate divider option depends on the criticality of the application, i.e., time-critical or space-critical. Based on that, one has to select the perfect alternative for the divider circuit or block in implementation. In the second section of this article, we discuss the various ways of stratifying different division algorithm classes for particular applications, which can be selected as either stand-alone or in combination with other classes to achieve the maximum efficiency in implementing the divider circuit or block. In later parts of an article, section three to section eight, we discuss the individual divider classes. Section nine discusses a large range of division algorithm implementations, followed by a comparative study in section ten.

II. DIVISION ALGORITHM BACKGROUND

All mathematical operations have been implemented using a digital platform, but it is still critical to implement the division operation. Researchers’ unceasing efforts in technological development have boosted computational complexities, which demand high-level systems performance. Nowadays, computers are ubiquitous in almost every field. A Field Programmable Gate Array (FPGA) is one of the outcomes of improved technology. It enables reprogrammable hardware, which reduces the hardware cost and implementation time.

3. Very high radix
4. Look-up table
5. Variable latency

Based on hardware architecture [8], [14], we can classify types of dividers as:

1. Serial or sequential type
2. Parallel type
3. Pipelined type

Based on performance [15], we can classify types of dividers as:

1. Slow type
2. Fast type

Based on execution [16], we can classify types of dividers as

1. Iterative subtraction type
2. redictive type

Various attempts have been made to study different division algorithms to state their quotient conversion logic, purpose, and design requirements [14], [17], covering some aspects of comparison. In the next section, we briefly discuss the comparative study of various classes along with different advantages and problems associated with them. There are five general classifications of division algorithms. Depending on the hardware architecture and accessing techniques, they can also be further characterized as serial, sequential, parallel, pipeline, slow, fast, iterative, and predictive classes, along with digit recurrence, functional iteration, very high radix, look-up table, and variable latency classes of division algorithm-based dividers.

III. DIGIT RECURRENCE CLASS (DRC)

The simplest and most commonly implemented division algorithm class is the digit recurring class due to its simple conversion logic. It is considered the oldest and pioneer class amongst all the division algorithms. Many surveys and research articles have been published based on these algorithm-based division circuits. At the beginning of the digital era, it was difficult to implement extensive algorithms due to the limited capabilities of programmable logic devices like FPGAs. Thus, the implementation of the DRC algorithm was preferred for commercial applications. The digit recurrence algorithm resembles the simple paper and pencil technique of division, as illustrated in Fig. 3 (a) above, in discussions on the process of a division operation, which works digit-by-digit and produces a quotient in sequence. It uses iterative type subtraction to calculate the quotient. This means the division is performed by repeated subtraction of the divisor from the dividend until the resultant quantity of subtraction is smaller than the divisor quantity. Quotient conversion logic is an iterative process of subtraction, which generates specific digits or bits of quotient at each iteration, from 1 to n digits or bits per iteration. In other words, the quotient is derived from a number of iterative subtractions that have been performed and is generated digit-by-digit in sequence, with its most significant bit first, like a paper and pencil algorithm [4], [5], [14]–[20]. The key point in using this type of divider is that it requires a combination of simple operations like addition,

shifting, multiplication, etc., shown in (1), and the remainder has to fulfill the requirement stated in (2) [4].

$$\text{Dividend} = (\text{Quotient} \times \text{Divisor}) + \text{Remainder} \quad (1)$$

$$0 \leq \text{Remainder} \leq \text{Divisor} \quad (2)$$

This class of division algorithm mainly covers three types of dividers

1. Restoring
2. Non-restoring
3. SRT (radix n)

Although it is easy and less critical, it has both merits and demerits. Being an easy and less complex conversion logic for the quotient is a merit, but it exhibits relatively higher latencies as a demerit. The long division algorithm is a good example of this. The speed of SRT-based dividers is mainly determined by the complexity of the quotient-digit selection logic. The division algorithm generally does not provide any finite result. It depends on the accuracy required to decide the length of quotient digits or bits. It has to use a quotient digit selection look-up table (QST) to enhance the quotient conversion time. It requires extra storage space either in ROM, programmable logic arrays (PAL), or combinational logic. Distinct sequential streams of digits represent the quotient and remainder, with the MSB digit or bit generated first in the quotient and remainder sequence. Many processors like Intel Pentium, HP PA 8000, and Sun UltraSPARC [20] initially implemented this concept.

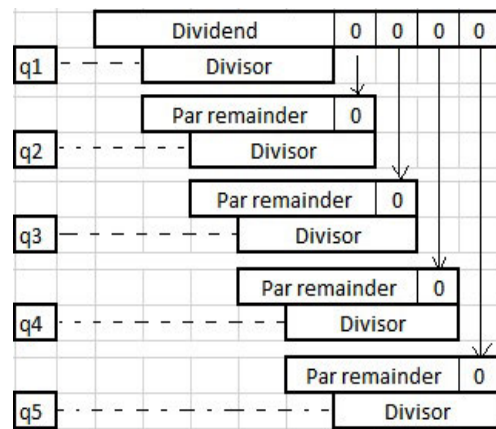


FIGURE 5. Long division algorithm of the digit recurrence class.

A. RESTORING ALGORITHM (RA)

The restoring algorithm has similarities with the long division method, which is also known as a paper and pencil algorithm in general, and is described in section I. Fig. 5 illustrates the long division algorithm of the digit recurrence divider class. In the case of standard long division, the algorithm's single quotient bit is calculated in each iteration by subtracting the divisor from the partial remainder generated in the previous iteration. In the case of the initial iteration, where the partial remainder is considered a dividend, the divisor's

initial iteration subtraction is performed from the dividend. The resultant partial remainder is considered for the next iteration. In each iteration, its divisor is checked against the shifted partial remainder of the previous iteration to verify the quotient bit for that particular iteration. If the divisor is found to be less than or equal to the shifted partial remainder, then the quotient bit for that iteration is considered to be one, else considered to be 0.

$$q_j = 0 \quad \text{if } 2R_{j-1} < D_r \quad (3)$$

$$q_j = 1 \quad \text{if } 2R_{j-1} \geq D_r \quad (4)$$

$$R_j = 2R_{j-1} - q_j \times D_r \quad (5)$$

Equations (25) to (27) represent conditions for the long division algorithm D_d to be the dividend, where D_r is the divisor, q_j is the quotient bit from the j^{th} iteration, and R_j is the partial remainder for the j^{th} iteration. No special case is required to test the maximum case in any iteration, which is nothing, but the initial dividend and value is equal to the divisor. Still, there is the possibility of losing a significant bit during the shifting process if the dividend is greater than the divisor, causing output error. Thus, an extra test case is required for checking the overflow state during the first iteration, and its last remainder is discarded; whereas, in restoring the algorithm at any moment, if the partial remainder value is other than positive or zero, then the divisor is restored by the subtraction result performed in that iteration.

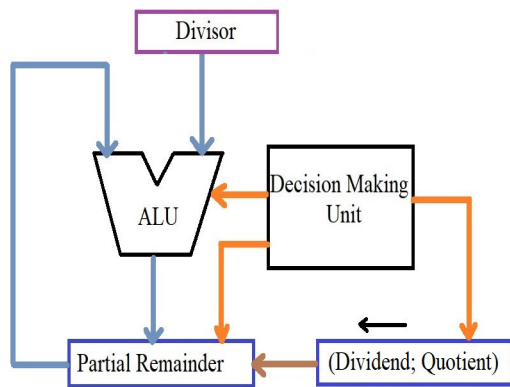


FIGURE 6. The restoring division algorithm of the digit recurrence class.

Fig. 6 illustrates the restoring division algorithm of the digit recurrence class. A non-redundant number system, which is also considered as a number system that doesn't use multiple bits to represent a single digit, is preferred to represent the quotient and remainder of the restoring algorithm-based dividers. To perform the division of the dividend number by the divisor number using a restoring algorithm, it is crucial to have a positive dividend and divisor; this is the essential requirement. Thus, the remainder and quotient values remain either positive or zero [4]. Its vital point for implementation is that it requires full-width comparisons to glean the new quotient digit. In every iteration of the algorithm, it performs shift, compare, add, and subtract operations. The steps to achieve this restoring algorithm are:

1. Select initial values for divisor (D_r), dividend (D_d), the partial remainder (R_j), and the number of bits (n) arranged in shift position to left, as indicated by the arrow sign shown in Fig. 6.
2. Subtract divisor (D_r) from the partial remainder (R_j), and the result is stored in the partial remainder (R_j).
3. Check for the most significant bit of the partial remainder (R_j); if 0, then the least significant bit of Q is set to 1; otherwise, the least significant bit of Q is set to 0, and the value of the partial remainder (R_j) is restored back to the value prior to the subtraction.
4. Reduce the value of n by one.
5. Continue iterations until we get a value of $n = 0$.
6. Lastly, the quotient (q_j) is obtained in the quotient dividend block.

Consider D_d is the dividend, D_r the divisor, q_j the quotient of the j^{th} iteration, and R_j the partial remainder. At the initial iteration, we can consider the dividend as partial remainder R_0 . The R_j and q_j values can be represented as the following equations:

$$R'_j = 2R_{j-1} - D_r \quad (6)$$

$$q_j = 0 \quad \text{if } R'_j < 0 \quad (7)$$

$$q_j = 1 \quad \text{if } R'_j \geq 0 \quad (8)$$

$$R_j = 2R_{j-1} \quad \text{if } q_j = 0 \quad (9)$$

$$R_j = R'_j \quad \text{if } q_j = 1 \quad (10)$$

B. NON-RESTORING ALGORITHM

This algorithm is very similar to that of the previously discussed restoring algorithm. One difference is that in a non-restoring algorithm, unlike the restoring algorithm, it is not required to restore the partial remainder if the subtraction goes negative. Similar to the restoring algorithm, in the non-restoring algorithm, we shift and subtract the divisor (D_r) depending on the value we get in the partial remainder, except that the range of the partial remainder (R_j) in the case of the non-restoring algorithm is $\{-D_r, D_r\}$. In a non-restoring algorithm, only one decision, either add or subtract, must be made per quotient bit q_n [4], [5], [15], [57]. There is no restoring step after the addition or subtraction decision is made to reduce the actions from the previously discussed restoring algorithm. The steps to perform this non-restoring algorithm are:

1. At the beginning, reset all values to zero.
2. Allot the corresponding values to the dividend (D_d), divisor (D_r), and the number of bits in the dividend (n).
3. Check for the sign bit of the partial remainder. For the first iteration, consider the sign positive, as the partial remainder value is set to an initial value of zero.
4. For the first iteration, subtract the divisor from the partial remainder.
5. If the result is negative, then shift the partial remainder left by one bit.
6. Add the divisor to the partial remainder.

7. After shifting the partial remainder one bit left in each iteration, the divisor is either subtracted from or added to the partial remainder, depending on the value of the previous iteration's sign bit.

It is mandatory to keep the partial remainder between the set of values $\{-D_r, +D_r\}$ [4], [5], [15]. Thus, we have to add or subtract in the next step. This implies testing when to add and when to subtract the divisor from the partial remainder. When the dividend is positive, the first iteration is always subtraction. Thus, the iteration may be set as $R_0 = 2D_d - D_r$; unlike the restoring algorithm, in the non-restoring algorithm, q_j ranges from -1 to $+1$ instead of 0 to 1 . In a non-restoring algorithm, it is required to maintain separate hardware for addition and subtraction for each iteration, causing overhead. The equations (11) to (14) represent the values of the partial remainder (R_j) and quotient (q_j).

$$q_j = -1 \quad \text{if } R_{j-1} < 0 \quad (11)$$

$$q_j = 1 \quad \text{if } R_{j-1} \geq 0 \quad (12)$$

$$R_j = 2R_{j-1} + D_r \quad \text{if } q_j = -1 \quad (13)$$

$$R_j = 2R_{j-1} - D_r \quad \text{if } q_j = +1 \quad (14)$$

The major drawback of this is that we need to maintain an extra sign bit to keep track of the sign and decide whether to perform addition or subtraction, which leads to deciding whether to perform addition or subtraction, leading to area and latency limitations when implementing this algorithm. Another minus point is that we need to maintain separate hardware to perform addition or subtraction. Thus, it suggests further optimization with 2's complement, in which 2's complement of D_r replaces $-D_r$ as (37) to (46) and Table. 1 summarize the basic points of comparison between restoring and non-restoring algorithm.

$$q_j = -1 \quad \text{if } R_{j-1} < 0 \quad (15)$$

$$q_j = 1 \quad \text{if } R_{j-1} \geq 0 \quad (16)$$

$$R_j = 2R_{j-1} + D_r \quad \text{if } q_j = -1 \quad (17)$$

$$R_j = 2R_{j-1} + \bar{D}_r + 1 \quad \text{if } q_j = +1 \quad (18)$$

C. SRT ALGORITHM (RADIX-N)

Digit recurrence algorithms are an enduring favourite for computer and electronic implementation. The SRT algorithm is one of the most popular of all the digit recurrence division algorithms to implement and one of the non-restoring digit recurrence algorithms. The primary application area of the SRT algorithm is in general-purpose processors, which are generally used for personal computers, FPGA systems, and ASIC processors. The SRT algorithm is named after the three individual researchers who individually proposed utilizing the 2's complement technique of shifting over zeros for the division to replace the range of the partial remainder in terms of reducing the resource requirements [15], [21]. As in the non-restoring algorithm, where the partial remainder is maintained in the $-D_r$ to $+D_r$ range, it requires an extra set of hardware to perform addition and subtraction. The SRT

TABLE 1. Comparison between restoring and non-restoring algorithm.

Restoring	Non-Restoring
It is similar to the long division method, which resembles a normal pencil and paper algorithm. It restores partial remainder while working.	It is similar to that of the restoring algorithm except restoring partial remainder.
When performing division on $2n$ bit number, it can require up to $2n+1$ adders.	As it eliminates the restoring cycle, it requires only n adders to perform division on the $2n$ bit number.
It doesn't allow -ve values of the partial remainder in between two consecutive iterations.	It allows +ve as well as -ve values of the partial remainder in between two consecutive iterations.
No error can be seen between consecutive iterations.	A small amount of error can be available during subsequent iterations.

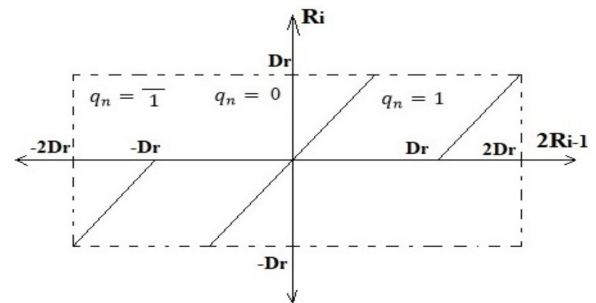


FIGURE 7. The radix-2 SRT algorithm.

algorithm implements 2's complement value of D_r instead of $-D_r$, which indeed provides shifting over zeros to eliminate the extra adder and subtractor [14], [15], [20], [21]. The following Fig. 7 and expressions illustrate the SRT algorithm for radix-2.

$$q_j = \bar{1} \quad \text{if } 2R_{j-1} < -D_r \quad (19)$$

$$q_j = 0 \quad \text{if } -D_r \leq 2R_{j-1} \leq D_r \quad (20)$$

$$q_j = 1 \quad \text{if } 2R_{j-1} \geq D_r \quad (21)$$

In the SRT algorithm, each quotient digit has one of the values $-m, -m + 1, \dots, -1, 0, +1, \dots, m-1, m$, where m is an integer [21], [58] such that (22) comprises k digits of radix- n as:

$$\frac{1}{2} (n - 1) \leq m \leq n - 1 \quad (22)$$

$$n = 2^b \quad \text{and } k = x/b \quad (23)$$

$$Q = \sum_{j=1}^k q_j n^{-j} \quad (24)$$

Quotient q is generated as a division of the dividend by a divisor of x bits significand, i.e., 4, 8, 16, 32, etc. The algorithm retires b bits of the quotient in each iteration. Thus, it is called a radix- n algorithm. Radix- n is typically

selected as a power of base 2. Such an algorithm performs k iterations to get the quotient. Thus, it shows the latency of k cycles, where the cycle time is considered as the maximum time to compute one iteration of the algorithm. This may or may not be the same as the clock time of the processor. This shows the algorithm's radix dependency, suggesting the higher the radix, the lower the latency time. The quotient digit is preliminarily guessed based on a few MSBs of the divisor and the partial remainder, rather than by computing. Thus, it requires a quotient digit selection and partial remainder generation in one iteration. Here the radix number n represents the trial subtractions performed while predicting the quotient [32], [33]. The IEEE has standardized some data formats commonly used for floating-point calculation, mainly named single and double-precision floating-point format with a significant 24 bits for single precision format and 53 for double precision format [5].

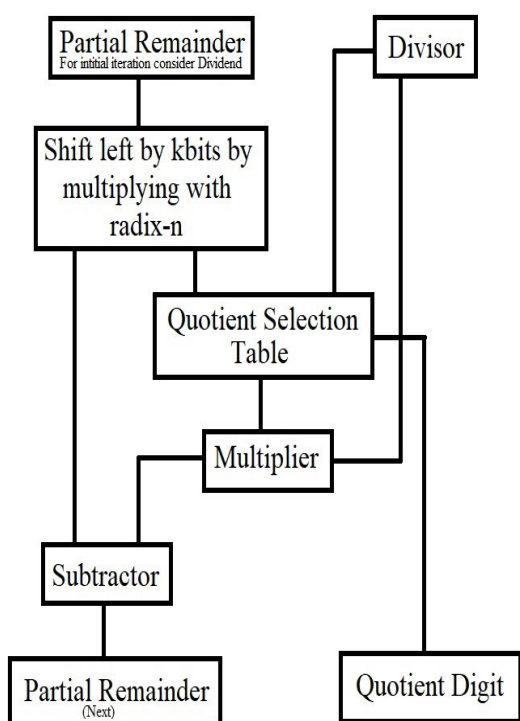


FIGURE 8. Block diagram of SRT algorithm.

Fig. 8 illustrates the SRT algorithm's block diagram performed at every quotient bit generation in every iteration. At the initial iteration, the partial remainder is considered as a dividend, and then it is multiplied by radix- n , which is represented as a shift left by k bits, as shown in Fig. 8. The resultant product is then given to the quotient selection table (QST) and the subtractor as one input. The divisor provides the second input for the quotient selection table. Now, based on a few MSBs of the product term and the divisor, it surmises the quotient digit for the next iteration. The second input to the subtractor is provided by the multiplier output, which works on the output of the quotient selection table and divisor to generate the next partial remainder. In the

next iteration, this partial remainder is used instead of the dividend. This will continue until all the quotient bits are revealed. In the last iteration, the generated partial remainder is considered as the final remainder. After the K^{th} iteration, the final quotient is achieved in redundant format, which shows that the resultant quotient can be represented in several formats, giving an alternative selection for the quotient digit in each digit position. Thus, it requires an extra subtractor to represent the final quotient in terms of a non-redundant number containing no negative digit. To achieve this, it is necessary to subtract the positionally weighted digit of the quotient from the positionally weighted positive digits. Carry out propagation is necessary to perform this subtraction once after the last iteration. The quotient selection is performed in the form of a redundant number system, which shows that a given position of the quotient digit requires the approximation of the divisor and partial remainder with a few MSB bits indicating the smaller error.

Meanwhile, the error in the guessed/predicted value of the quotient and the partial remainder relates directly to the number of unexamined bits from the divisor and partial remainder. It is expected that smaller errors possibly be resolved by the less significant bits of the quotient. Equation (22) suggests the range of maximum digits to consider, which is represented by m . If we select a lower range where m is equal to the value $(n-1) / 2$, this shows lower redundancy, and m equal to the value $(n-1)$ shows maximum redundancy. Higher redundancy eases the quotient selection logic design and requires fewer bits from the partial remainder to be examined.

On the contrary, this requires more multipliers of the divisor to be formed; this will make it necessary to pre-compute the values of the multipliers and also requires extra space to include with the actual algorithm along with the quotient selection table. From this implementation point of view, the available choices with specific components will contribute to the cost, area, and, ultimately, the algorithm's performance. The trade-off between these components will lead to different application choices, from less critical to critical, and affect the time-cost requirements. The components with choices to be made are mainly the radix, quotient representation, and partial remainder representation [14], [23], [58], [79].

1) CHOICE OF RADIX

In general, the radix is termed as a base number, which is primitive and from which we can produce other numbers in connection, which can be termed as the number system. It is also termed as the fundamental number of any system. In the case of the SRT algorithm, it considers the power of 2 for selecting different radix types. The main reason to consider the power of 2 here is that the product of the partial remainder and radix can be presented as a shifting operation, which makes for the easier design of the hardware. Here radix- n indicates how many quotient bits will be revealed and, for that, how many subtraction stages are required. Thus, increasing the radix will increase the quotient bits revealed in one

iteration, causing a reduction of the total iterations required to get the ultimate quotient. As radix-2 retires one quotient bit per iteration and radix-4 retires 2 bits per iteration, this reduces the latency.

On the contrary, it increases the complexity of the logic for quotient digit selection. In practice, the iterations are reduced due to an increase in the radix, but this also increases the criticality in the quotient digit selection logic, requiring a longer look-up table to be implemented. Thus, the time required to access the quotient selection table will increase with the radix increase, possibly increasing the total time required to compute the quotient bit. So, the total time required to compute n quotient bits is not reduced as per the calculations. In general, the radix- n SRT algorithm is implemented serially so that a single look-up table can be used for all iterations. Thus, the maximum hardware implementations are restricted to radix-4 SRT [4]. Along with this, more multipliers need to be formed for an increased radix, requiring a greater area. Thus, these two factors adversely affect the advantage of an increased radix, making lower radix values preferable for implementation, which also introduces some error in the predicted value and the exact value of the quotient, which can be resolved at the least significant bits in the last iteration.

2) CHOICE OF QUOTIENT DIGIT SET

In digit recurrence algorithms, it is possible to decide digit ranges; in short, to decide the value of a digit among the given set of possible values. To improve the algorithm’s speed and performance, we use symmetric consecutive digits with signed bits with a maximum possible value of m , and the number of digit values must contain higher than N consecutive integer values, including value zero, i.e. $-m, -m + 1 \dots -1, 0, +1 \dots m-1, m$. Digit value must be valid for $m \geq n/2$ condition to make the digit range redundant. The redundancy factor ρ , is responsible for the redundancy of digit range [69]–[74], [76], which can be expressed as (25)

$$\rho = m / (n - 1) \quad \text{and} \quad \rho > 1/2 \quad (25)$$

When the value of m is $n/2$, digit range is minimal redundant, and when m is $(n-1)$, the digit range is maximum redundant. Once the redundancy factor ρ is selected, we can perform quotient selection logic. Thus, while performing quotient digit selection from the digit range, we have to consider the containment condition defined by the sectional interval between two consecutive redundant digit values in the digit range. We can represent the containment condition as a region covered by conditions given in (26), where H_k and L_k stand for higher and lower cut off, which can be represented as a line with slope $\rho + k$ and $-\rho + k$ [14], [21], [58], [62], [64]–[68].

$$H_k = (\rho + k) D_r \quad \text{and} \quad L_k = (-\rho + k) D_r \quad (26)$$

To improve performance, we consider using a redundant digit set, which allows us to select the quotient digit based on the partial remainder. This introduces a small error, which can be rectified in a later iteration: e.g. the radix-2 digit set is

$(-1, 0, 1)$ and for radix-4 there are two possibilities, a minimal set having $(-2, -1, 0, 1, 2)$ and a maximum set $(-3, -2, -1, 0, 1, 2, 3)$ [15], [32], [35], [70], [72]–[76]. A greater possible value for the quotient bit leads to simplifying the logic for quotient digit selection, but at the same time, it will make a more complex product of the partial remainder and divisor, which may require more multipliers, causing an area increase.

The most critical part of divider performance is how efficiently implemented quotient selection logic. If we use redundant digit value representation for remainder digits, then we will not be able to derive the exact value of partial remainder or residue, which will cause uncertainty in selecting an exact value for the next quotient digit. Thus We have to use redundant digit value range representation for selecting quotient digit. When we use redundant digit value range representation for selecting quotient digit, it is not important to know the exact value of partial remainder or residue, but it must be required to know, as shown in Fig. 9, the exact location in which sectional interval range of partial remainder - divisor graph it will fall. The realization of quotient digit selection logic is performed by approximating partial remainder and divisor. The quotient selection logic’s complexity depends upon how many bits of partial remainder and divisor are utilized. A separate look-up table is performed, which contains all possible values of the selection logic. In the generalized look-up table method, we utilize selection constants. We perform sectionizing the complete divisor range into equal intervals (D_{rj}, D_{rj+1}) expressed as

$$D_{r1} = 1/2, \quad D_{rj+1} = D_{rj} + 2^{-\delta} \quad (27)$$

Two consecutive sections share an overlapping region, as shown in Fig. 9. Extra attention needed to be given while deciding logic to select quotient digit in this overlapping region. The regions are indicated by the most significant bits

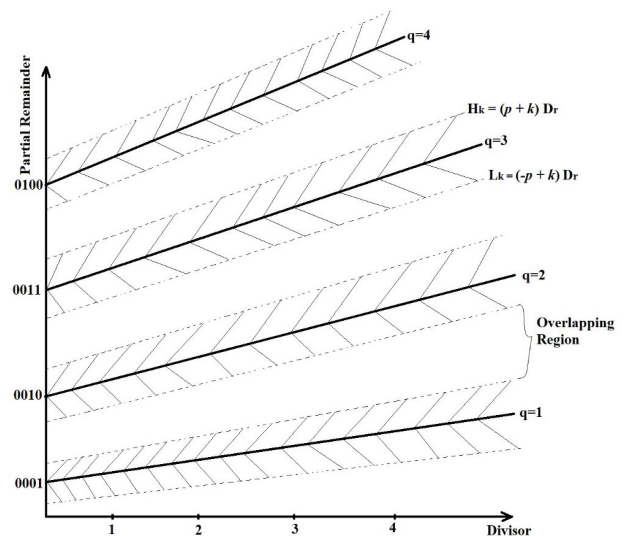


FIGURE 9. Partial remainder – Divisor (PD) graph.

of the divisor, which are used for selection logic based on the selection constant $S_k(i)$ is given as

$$q_{j+1} = k \quad \text{if } s_k(i) \leq nr_j \leq s_{k+1}(i) - n^{-r} \quad (28)$$

then the range of selection constant $s_k(i)$ for a given value of k forms a series of steps that connect the overlap region. Higher redundancy factor cause wider steps and requires less divisor and partial remainder bits. However, an increase in the radix directly influences the quotient digit selection logic's complexity and, ultimately, the look-up table. The vital problem associated with the SRT algorithm is predicting quotient digit in the overlapping region caused by the same region corresponding to different coefficients. Quotient digit value has to be one which can either be $q = q_j$ or $q = q_{j+1}$ depending on the selection logic derived by the divisor and partial remainder [14], [35], [51], [58], [60]–[62], [64]–[68]. The step function is not constant for all overlapping regions. Depending upon what radix is used, causing the more divisor-sectional regions, increasing step function in the higher radix. This small looking problem can cause a magnificent loss in practical implementation in terms of cost and time and lead to total system failure. The most famous example of this problem is Intel's Pentium processor flaw in the Floating-point divider, which was design based on the SRT algorithm [32], [61], [65], [69], [71], [76], [78]. A potential problem in overlapping regions costs USD 475 million to Intel to replace the faulty Pentium processor chip [65].

3) CHOICE OF REMAINDER REPRESENTATION

There are two options available with the SRT algorithm to represent its partial remainder and remainder, which are the redundant and non-redundant format. The conventional 2's complement is an example of the non-redundant form, while the carry-save two's complement is an example of a redundant form. When we consider the non-redundant form, then subtraction is required to find the partial remainder required to implement the carry propagated full-width adder. When we use the redundant form, then subtraction can be performed by carry-save adders, but this complicates the quotient digit selection logic as it is dependent on the shifted partial remainder value. A summary of the SRT algorithm is given in Table 2.

4) SRT ALGORITHM PERFORMANCE IMPROVEMENT TECHNIQUES

As we have stated earlier, the SRT algorithm has been very popular from the very beginning. Thus many attempts have been made to improve the performance of the traditional SRT algorithm. As we have discussed in earlier sections on different parameters and how they affect the traditional SRT algorithm's performance, many techniques have been claimed to improve the traditional SRT algorithm's performance, some of which are discussed in [34]–[44]. Some of the performance-improving techniques like simple staging, overlapping execution, overlapping quotient selection, overlapping partial remainder computation, range reduction, operand

TABLE 2. Summary of SRT algorithm.

Pros	Cons
Simplicity in low radix implementation due to linear convergence.	Linear convergence of quotient bit makes it considerably slow with large bit size operands (input).
Availability of final remainder and quotient at the end of the computation.	Required normalized operands.
Use redundant digit set representation for input operands; this allows a valid quotient digit to be selected from just estimating the current partial remainder and reducing it to execute redundantly.	The possibility to select more than one quotient digit values during quotient digit selection due to the availability of overlapped region in quotient bit selection logic makes it critical and cause a big problem in actual working.
Avoids carrying propagation during reduction.	Needs critical attention for designing quotient bit selection logic for higher radix implementations.
Uses symmetric signed digit consecutive integers with maximum digit value m .	It requires a multipliers range of power of 2. If not, then it requires extra hardware for addition circuit.
No requirement of prescaling for operands.	The step function is constant for all overlapping regions. It increases with an increase in radix, causing it critical to work with a higher radix level where the number of overlapped regions and quotient digit selection values are also more.
The conversion speed depends upon the number of cycles required to finish the computation. Speed of conversion increases with increase radix.	The quotient digit selection logic look-up table grows quadratically with an increasing radix, containing thousands to millions of entries.
Few MSB's of the divisor and partial remainder are required for generating quotient digit selection logic look-up table.	*****

scaling, and circuit effects are important and discussed in the later sections.

a: SIMPLE STAGING

Cascading is the method used to connect two blocks of circuits back-to-back, suggesting that one circuit's output is connected to another circuit's input. In terms of the SRT algorithm, if we connect two low radix divider circuits back-to-back in a cascaded fashion, it can work as a higher radix divider as one unit, as shown in Fig. 10.

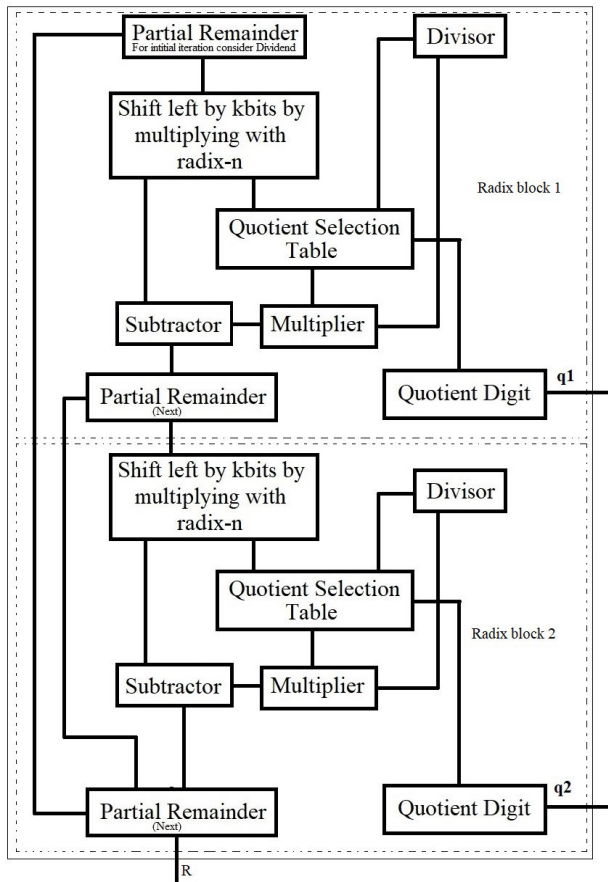


FIGURE 10. Block diagram of cascaded implementation of SRT algorithm.

Cascading multiple lower-order radix dividers together will contribute to a higher radix divider, but at the cost of higher area requirements than the usual one unit of a high radix divider. The key point in using multiple low radix blocks together is to use them at a much higher clocking frequency than the system clock frequency; likewise, we are able to work out multiple blocks in one system clock cycle. It is possible to arrange multiple low radix dividers to completely determine all the quotient bits in one system clock cycle. The major drawback of this may be an enormous amount of area and having an unacceptably low cycle time. HP PA-7100 and AMD 29050 microprocessors are examples of two radix-4 clocking faster than the system clock to perform radix-16 work in every machine cycle [14].

b: CIRCUIT FAMILY EFFECT

The study shows that the two circuits built using the same logic family of digital circuits cause similar delays. If the same circuits are implemented in the different logic family of the digital circuit, this shows visible changes in the circuit's performance, either worse or better. The study presented in [32], [34] shows that many circuit-level implementations of the SRT algorithm yield different performance depending on the choice of base architecture and the choice of

radix-2 or radix-4. When performed, implementation in CMOS and dual-rail domino circuits provide a 1.5 to 1.7 times speedup performance.

c: OVERLAPPING / PIPELINE EXECUTION

A divider circuit is a very complex operation formed by connecting different components sequentially and logically, which makes it possible to overlap some of the operations of components to execute them together in the same cycle. This ultimately leads to a pipeline structure of the components and reduces the execution cycles [14], [43].

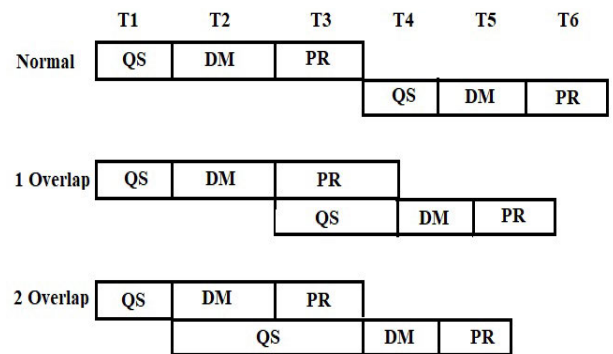


FIGURE 11. The conditions of execution.

Fig. 11 illustrates the three different conditions of execution of the SRT algorithm. In short, the SRT algorithm has three components, which execute their work after finishing the previous component's execution, as shown in the normal form of SRT algorithm execution. In the normal form of execution, the second iteration starts after completion of the first iteration, indicating that the next stage's quotient selection (QS) is dependent on the partial remainder (PR) generated in the previous iteration. The execution depends on the partial remainder execution time in the one overlap form, suggesting that overlapping quotient selection execution depends on the partial remainder execution time. In the case of 2 overlaps, execution is dependent on the quotient selection execution time, which indicates a pipelining quotient selection execution along with the divisor multiplier (DM) and partial remainder execution time. The partial remainder dependent pipelined form of execution is performed when a redundant format is used to represent the partial remainder, whereas the quotient selection execution dependent pipeline is suitable when a non-redundant format is used to represent the partial remainder.

D. SVOBODA ALGORITHM (GSA)

SRT is the most implemented digit recurrence algorithm, which works on the principle of developing a quotient digit selection logic based on a few MSB's of the divisor, and the partial remainder. SRT does not require prescaled operands, but it worked on the normalized operands. In 1963, Svoboda came up with a radix-n digit recurrence division algorithm based on the only partial remainder. Unlike the SRT digit

recurrence algorithm, it considers quotient digit selection logic based on remainder's MSBs [10], [59], [80], [95]–[100]. Svoboda division algorithm, also known as generalized Svoboda division algorithm or simply GSA. Svoboda division algorithm requires inputs to be in prescaled form, near to 1. Thus, it can be represented as $(1 + e_r)$, where e_r is a small positive fractional value $e_r < 1/n$ and n is the radix. We can describe the Svoboda digit recurrence algorithm in simple steps [95], [96] as

- In the first stage, consider normalized inputs. If not, convert it in normalized form and prescale operands to represent $(1 + e_r)$, i.e., near to 1.
- In the second stage, the actual iteration process will start. At each iteration j , the quotient digit of that iteration q_{j+1} is multiplied by a small positive fractional value e_r and subtracted from the partial quotient q_j . The resulting partial quotient bit is considered for examination.
- If q_j results in $-ve$, it indicates overshooting, to compensate overshooting by adding/subtracting e_r and performing right shift operation by $j-1$ places depending on the last step was subtraction/addition.
- After i^{th} iteration, left i^{th} digits of the partial remainder are considered as quotient digits, and the rest of the digits are considered remainder.

Even though the Svoboda digit recurrence algorithm requires only remainder MSB digits to estimate quotient digits, there are certain limitations [59], [80], [95], [99] to Svoboda implementation

- It requires prescaled inputs in a particular range near to 1, causing additional clock cycles.
- Extra two multiplications are needed if operands are not in prescaled form.
- Possible overflow due to overcompensation causing to select quotient digit from out of the remainder digit range.
- It is applicable above $n > \text{radix } 4$.

E. SVOBODA-TUNG ALGORITHM (STA)

Later Tung [59], [95], [97], [98] investigated the possibility of the Svoboda algorithm implementation with the signed digit number system, whereas the generalized Svoboda division algorithm is implemented on redundant digit representation. Tung Implementation of Svoboda algorithm known as Svoboda-Tung (ST) algorithm. Svoboda-Tung (ST) algorithm also exhibits the same drawbacks as that of the Svoboda algorithm mentioned above. Along with that, Tung has exploited the carry propagation free property of the signed digit number system and the simplicity of quotient digit selection logic [95]. Later in 1991, Burgess [95] has implemented Svoboda-Tung (ST) algorithm with a slight change. In Burgess implementation, they have considered two MSB's of partial quotient instead of one MSB to determine quotient digit. Upon the worst condition of overshoot, unlike Svoboda-Tung (ST) algorithm, here it gives several possibilities which are summarized as $\{00, 01, 0\bar{1}, 10, 11, 1\bar{1}, \bar{1}0, \bar{1}1, \bar{1}\bar{1}\}$ to perform different

controlling operations defined for all the alternatives given in the range, like not operate when MSB value is 00, 01, 0 $\bar{1}$, Subtract e_r when MSB value is 10, 11, add e_r when MSB value is $\bar{1}0, \bar{1}\bar{1}$, rewrite 01 when MSB value is $1\bar{1}$ and rewrite $0\bar{1}$ when the MSB value is $\bar{1}1$. Where sign digit range is given as

$$\text{Range} = \{0, \pm 1, \dots, \pm m\} \quad (29)$$

$$\text{Boundry limit} = \{n/2 + 1 \leq m \leq n - 1\} \quad (30)$$

In this, m in (29) is considered the maximum digit value in the balanced signed digit range, which could be selected for the quotient digit, and n in (30) is the radix of dividers. The arithmetic limit for the partial remainder is given as (31), and the valid range of divisor (D_r) is given as (32)

$$\{-m/n - 1 < R_j < m/n - 1\} \quad (31)$$

$$\left\{ \frac{mn}{(m+1)(n-1)} < D_r < \frac{m(n-2)}{(n-1)(m-1)} \right\} \quad (32)$$

F. NEW SVOBODA-TUNG ALGORITHM (NSTA)

To overcome the basic drawbacks of the Svoboda-Tung (ST) algorithm without losing up any of the benefits is possible by incorporating the following updates in the actual Svoboda-Tung (ST) algorithm [59], [97], [98], [100], signed digit range is given as

$$\text{Range} = \{0, \pm 1, \dots, \pm m\} \quad (33)$$

$$\text{Boundry limit} = \{n/2 + 1 \leq m \leq n - 1\} \quad (34)$$

In this, m in (33) is considered the maximum digit value in the balanced signed digit range, which could be selected for the partial remainder R_j along with the signed binary digit (SBD) range given in (35) and n in (34) is the radix of dividers. The valid range of divisor (D_r) is given as (36)

$$\text{SBD} = \{-1 \leq m \leq 1\} \quad (35)$$

$$D_r \text{ range} = \{0, 1, \dots, n - 1\} \quad (36)$$

This arrangement allows for addition /subtraction with carry propagation up to one left position. The second drawback of ST, i.e., overshoot due to compensation, is avoided by implementing the alternative method of recoding two MSB's of the partial remainder with alternate consecutive positions causing to follow and keep the partial remainder in bounded condition

$$\text{Boundry limit} = \{-m/n - 1 < R_{j+1} < m/n - 1\} \quad (37)$$

IV. VERY HIGH RADIX CLASS

Very high radix class algorithms are similar to non-restoring digit recurrence class algorithms. In short, we can differentiate the Simple SRT algorithm and high radix algorithm based on the number of quotient bits retired in one iteration. Generally, a divider retiring more than 10 quotient digits in one iteration qualifies as a very high radix algorithm. These very high radix algorithms show different hardware and logic arrangements for quotient selection and partial remainder generation

than SRT-based radix $-n$ algorithms. The main difference between the SRT and high radix algorithm is that it has a more complex divisor multiple process and quotient-digit selection hardware, which increases the cycle time and area. Similar to the low radix SRT algorithm, a very high radix algorithm also uses a look-up table, but the size and complexity are greater. The high radix algorithm proposed by Wong and Flynn [22] requires hardware with at least one look-up table of size $2^{(m-1)}m$ bits. Three multipliers are required, with a carrying assimilation multiplier of size $(m+1) \times n$ for the divisor's initial multiplications, a carry-save multiplier of size $(m+1) \times m$ is used to compute the quotient segments. The look-up table has $m = 11$, i.e., $2^{(11-1)} = 1024$ entries, each 11 bits wide, so in total, 11K bits are required in the look-up table with the slower implementation of the algorithm. In contrast, the fast implementation of the algorithm requires a look-up table with 736K bits. The high radix algorithm proposed by Lang and Nannarelli [45] shows the construction of a radix- 2^K divider for implementing a radix-10 divider whose quotient digit is decomposed into two parts, one in radix-5 and the other in radix-2. In radix-5, the quotient digit is represented as values $\{-2, -1, 0, 1, 2\}$, requiring three multipliers. Radix-2 is used to perform division on the most significant slice. It uses an estimation technique in the quotient selection component, which requires the use of a redundant digit format.

The Cyrix 83D87 arithmetic co-processor utilizes a short reciprocal algorithm similar to the accurate quotient approximation method to obtain a radix-2 17 divider [14]. The Cyrix divider has a single 18×69 rectangular multiplier with an additional adder port to perform a fused multiply/add. Therefore, it can also act as a 19×69 multiplier. Although the high radix division algorithm works with a scaling dividend and divisor by correct initial approximation of the reciprocal followed by quotient selection logic with a multiplier and subtraction, it exhibits the basic SRT properties radix- n algorithm. It uses the reciprocal approximation to investigate the correct quotient bit based on the formatting scaling factor based on the look-up table, instead of the look-up table only. It requires post-correction and rounding off if needed, with final sign detection. In short, we can say that the high radix dividers are the same as that of SRT based radix dividers with a basic difference of increased complexity and criticality in quotient digit selection techniques. Higher complexity and criticality in SRT based radix divider is not the only way to implement high radix dividers, as early we said that a combination of two or more alternatives together could solve this problem for high radix implementation. Many research works are going on all around the world to provide different aspects for high radix dividers. Use of different look-up tables along with quotient digit selection logic look-up table [66], [80], [83], speculating quotient digit and using arithmetic functions to multiplicative iterations rather than subtractive iterations [51], prescaling operands [88]–[93], using Fourier division [86], [87], using alternative digit codes like BCD digits instead of decimal and basic binary digits [81], cascading

multiple stages of lower radix dividers [77], overlapping two or more stages of low radix [32], [67], a truncated schema of exact cell binary shifted adder array [68], [82], [85], on-line serial and pipelined operand division [84], parallel implementation of the low radix dividers [94], array implementation [6], these are some of the possible ways applicable for high radix dividers.

V. LOOK-UP TABLE CLASS

A look-up table class algorithm can be utilized along with functional iterative class and high radix algorithms. For lower precision applications like consumer electronics, it can be used to avoid subsequent use of the algorithm. Look-up tables can be used to hold the values of pre-computed values for the quotient bit finalizing technique, standard values, etc. SRT radix- n is the best example of a look-up table class division algorithm. The approximation can be achieved by a look-up table that can provide a faster option at the cost of an increased area. As the number of bits increases, the look-up table area requirements also increase. Direct approximation and linear approximation require the use of the look-up table for the initial approximation value. In direct approximation processes, it is expected to prepare a look-up table containing the exact value of the approximation of the reciprocal function directly at every stage separately. In this case, the table is formed by the entries of the reciprocal of the midpoint and successor in the range $1, b_1, b_2 \dots b_k$. The recent upcoming stated in [46] about the bipartite reciprocal table, which can be used for approximation in dividers. It uses two separate look-up tables for positive and negative values. The table forms result in a redundant format which needs further conversion using multipliers. In the case of linear approximation, the look-up table uses some polynomial approximation, which can be expressed as a truncated series as in (38).

$$P(a) = X_0 + X_1a + X_2a^2 + X_3a^3 \quad (38)$$

The initial order coefficients X_0 and X_1 are stored in the look-up table, followed by multiplication and addition. The absolute error in the final iteration values depends on the initial approximation. In the case of a linear and direct approximation look-up table, it depends on the trade-off between the j number of iterations and the n^{th} number of bits provided to the look-up table. The look-up table class is hybrid, in which look-up tables are utilized to improve different classes of algorithms; e.g., the look-up table can be used in the SRT algorithm to store the quotient bit selection table and in functional iteration class algorithms to store the elements required for initial approximation, which ultimately reduces the absolute error. Moreover, one more type of algorithm is discussed in [16], explaining the use of the look-up table for storing and utilizing pre-computed values to perform the division operation. In the algorithm, it first scales down the denominator in the range of 0.5 to 1; then it refers to the pre-computed value for the reciprocal of a scaled-down divisor to multiply with the numerator to get the quotient bit. The drawback of this is that it generates an absolute error.

VI. FUNCTIONAL ITERATION CLASS

Unlike linear convergence algorithms where a single digit of quotient is calculated in every iteration, a functional iteration divider computes the quotient of division by estimation; thus, it can give more than one digit of the quotient in one iteration. This division method is based on the use of multiplication instead of subtraction, which ultimately reduces the iterations and can generate multiple quotient digits in one iteration with low latency at the cost of the accuracy of the ultimate result. The implementation of multiplication for conversion requires a greater area, and for that purpose, it is implemented with small size multipliers. The use of multiplication for functional iteration dividers makes it more complex than simple digit recurrence dividers. This type of divider has a major drawback of the inaccuracy of the quotient result of direct rounding off approximate solution values rather than infinite precise values. Functional iteration based algorithm performs division effectively but fails to give exact results every time. They employ rounding off methods while converging towards quotient, which allows keeping some rounding off error [110]. The standard of rounding off includes four techniques named RN, RZ, RM, and RP, of which RN is unbiased rounding to the nearest method, which performs rounding even if in a Tie case. Functional iteration dividers work on the series expansion phenomenon, some of which is shown in the [14], [47]:

1. Newton–Raphson algorithm (NRA)
2. Goldschmidt algorithm (GSA)
3. Series expansion algorithm (SEA)
4. Taylor series algorithm (TSEA)

A. NEWTON–RAPHSON ALGORITHM (NRA)

As shown in (39), it is considered possible to express the result of the division process as a single term of a product of the dividend and anti-divisor (reciprocal). To compute the anti-divisor in the Newton–Raphson algorithm depends on selecting the priming function, which points out its root at the anti-divisor [14], which generally has many values. Based on which root is selected, the quotient convergences accuracy will vary, causing an error in the division and generating overhead if the root selected is over the true quotient as indicated by (43). This indicates that the accuracy can be improved by first selecting the proper root, which can cause a reduction of latency. Thus, latency and error in the convergence are directly dependent on the root selected at the beginning of the convergence [101]. The same method is used in IBM 360/91 and Astronautics ZS-1 [24], [25].

$$Q = D_d/D_r = p \times (q)^{-1} \tag{39}$$

$$f(X) = 1/X - q^{-1} = 0 \tag{40}$$

$$X_{i+1} = X_i - \frac{f(X_i)}{f'(X_i)} \tag{41}$$

$$X_{i+1} = X_i - \frac{(1/X_i - q^{-1})}{-1/X_i^2} = X_i \times (2 - q^{-1} \times X_i) \tag{42}$$

$$\epsilon_{i+1} = \epsilon_i^2 (q^{-1}) \tag{43}$$

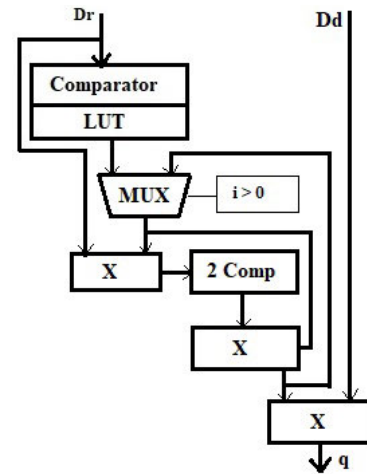


FIGURE 12. The block diagram of Newton–Raphson algorithm implementation.

Fig. 12 above illustrates the block diagram of the Newton–Raphson algorithm implementation for the division. After applying the dividend and divisor, the Newton–Raphson architecture starts with the first approximation to find the anti-divisor ($x_1 = D_r^{-1}$), i.e., the reciprocal or anti-divisor, and store it in LUTs. Multiplexers make a choice of selecting the initial approximation, and then the multiplier is used to generate product term D_{r0} , and the D_{r1} result is fed to 2's complement block for $(2-p)$ calculation, and the result is fed to the second multiplier, which computes the value of the new approximation $x_2 = D_r^{-1}(2-p)$. This new approximation is utilized to find a new partial remainder, which is required to calculate the next approximation. After the last iteration output of the second multiplier is fed to the last multiplier to find the final value for the final approximation, it shows that each iteration works on refining the anti-divisor (reciprocal), and after n iterations, the quotient approximation is performed by the last multiplier. Thus, we can divide the Newton–Raphson algorithm into three parts, namely initial estimation of the quotient approximation, the iterative process to approach nearest to the final value, and convergence to the anti-divisor, i.e., the reciprocal. A major drawback is that it requires a large gate count, and with an increase in the iterations, it increases to an enormous amount, which is not practically possible to implement.

B. SERIES EXPANSION ALGORITHM

Another known method of functional iteration is the series expansion method, in which the series can represent the root of the anti-divisor or reciprocal, which can be used in the iterations for rounding off. As per (44), series expansion is equivalent to the Newton–Raphson iteration for value $X_0 = 1$. Unlike Newton–Raphson iteration, which implements convergence of the anti-divisor followed by multiplication with the dividend, in series expansion, the iteration performs pre-scaling of the dividend and divisor by series approximation or rounding off and then performs series convergence.

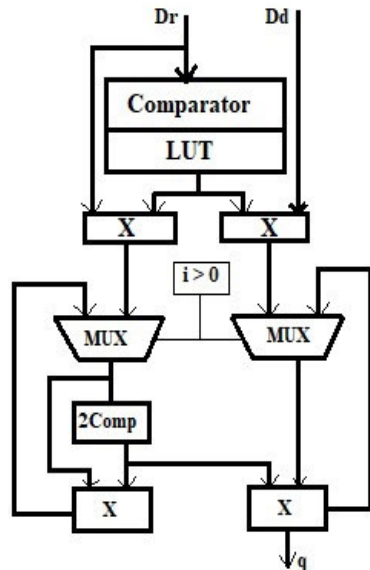


FIGURE 13. The block diagram of Goldschmidt algorithm implementation.

Thus, it shows the possibility of the use of pipeline or parallel hardware architecture. For performing series expansion, a Taylor series is used for function $g(y)$ at point a, p . Very often, this series expansion method is named the Goldschmidt algorithm [48].

$$g(y) = g(p) + (y - p)g'(p) + \frac{(y - p)^2}{2!}g''(p) + \dots \quad (44)$$

$$q = a/b = a \times g(y) \quad (45)$$

Fig. 13 above illustrates the block diagram of Goldschmidt algorithm implementation. Similar to the Newton–Raphson algorithm, the Goldschmidt algorithm also uses initial approximation $g_1 = D_r^{-1}$ stored in LUTs. The next step computes quotient approximation $q_1 = g_1 * D_d$ and error $e_1 = g_1 * D_r$ in the initial iteration; parallel multipliers consider the value of q_1 and e_1 to calculate the value of g_2 . Later parallel multipliers calculate the new quotient approximation and error. This means that this algorithm generates a new quotient approximation at each iteration, unlike the Newton–Raphson algorithm.

C. GOLDSCHMIDT DIVISION ALGORITHM (GDA)

Goldschmidt division algorithm (GDA) is one of the convergence-based algorithms used for performing division, similar to that of the Newton-Rapson algorithm [53], [105]–[109]. Like the Newton-Rapson algorithm, GDA also offers quadratic convergence of quotient, but there is a difference between them. Unlike the Newton-Rapson algorithm, which first calculates anti-divisor and then multiplies with dividend, the Goldschmidt division algorithm multiplies both dividend and divisor by anti-divisor [53], [109]. Contrary to subtractive iteration based algorithms, convergence based multiplicative iteration algorithms perform interaction between adder output and control logic only after

multiplication [109]. Goldschmidt division algorithm originates from the Taylor-Maclaurin series of $1/(x + 1)$ [109]. The basic operation of the Goldschmidt division algorithm can be expressed as [53], [105], [106], [108]

$$D_d/D_r = N/D = A/B \quad (46)$$

$$x_{n+1} = x_n(2 - y_n) = x_n r_n \quad (47)$$

$$y_{n+1} = y_n(2 - y_n) = y_n r_n \quad (48)$$

where,

$$x_0 = D_d * LUT(1/D_r) \quad (49)$$

$$y_0 = D_r * LUT(1/D_r) \quad (50)$$

$$LUT(1/D_r) = LUT(f(t)) \quad (51)$$

Equation (47) and (48) shows that x_n, y_n are bound to 2, and the value used for multiplication is always calculated by subtracting the divisor’s current value from 2. The division boundary condition is set to $\{1/2 < D_d/D_r < 1\}$. This algorithm’s major drawback is that it does not provide the remainder, making it useful only for the floating-point division [109]. First multiplication required for finding out values of x_n , and y_n requires full precision. Another drawback, 1’s complement can be used instead of $(2 - y_n)$ to avoid carry propagation delay, but it adds a new approximation error in each iteration. To overcome this problem, one basic observation comes in handy that when y_n reaches to 1 then $(2 - y_n)$ also reaches to 1, which can be advantageous for implementation by reducing area and performance at a time [109]. Later Harrison and Markstein found that the actual Goldschmidt division algorithm can be expressed as recursive equations that use multiplication and square operations in each iteration [105], [106], [108]. Such type of applications of Goldschmidt division algorithm is termed as modified Goldschmidt division algorithm and useful for software library implementation [109].

D. TAYLOR SERIES ALGORITHM (TSA)

Extended latency in dividers is seen because of the use of the 1st order Newton-Rapson algorithm and binomial expansion based Goldschmidt algorithm because of the major issue regarding reusing multiplier in between two subsequent operations [102], [104]. The next operations have to wait in subsequent operations until the preoccupied multiplier gets free from the previous operation. Taylor series expansion is also a multiplicative iteration division algorithm like Newton-Rapson and Goldschmidt algorithm. As we previously discussed in multiplicative algorithms, the precision depends upon the closeness with anti-divisor (reciprocal) estimation. Thus Taylor series expansion is used to calculate accurate anti-divisor (reciprocal) to reduce the error in the least important bits of quotient precision. Taylor series expansion dividers work in two stages [53], [102], [103]

- In the first stage, after providing both operands, it performs an estimation of anti-divisor (reciprocal).

- In the second stage, partial remainder and quotient are reformed during multiplicative iterations of Taylor series expansion until expected precision is achieved.

$$q = D_d/D_r \quad \text{and} \quad X_0 = 1/D_r \quad (52)$$

$$q = D_d X_0 \left\{ \begin{array}{l} 1 + (1 - D_r X_0) + (1 - D_r X_0)^2 \\ + (1 - D_r X_0)^3 \end{array} \right\} \quad (53)$$

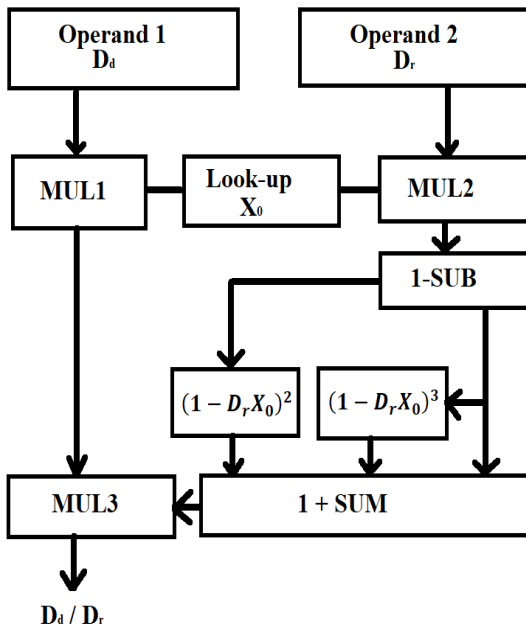


FIGURE 14. Operational block diagram of Taylor series algorithm.

Taylor series expansion implementation provides a parallel powering section that computes high order terms faster with minimal extension to hardware overhead. An operational diagram is shown in Fig. 14. It is used in IBM RS/6000 and AMD K7 processors [30], [104]. Even though the Taylor series expansion gives a better anti-divisor value, the huge amount of operational multiplication load, causing more power and area utilization [30]. Later, Liu *et al.* [30] presented a hybrid algorithm formed by combining prescaling, series expansion, and Taylor series expansion for a different purpose in dividers implementation. In their proposed structure, in the first stage, prescaling is used to prescale input operands to keep the divisor in the proper range. In the second stage, the series expansion algorithm performs an accurate anti-divisor prediction, which can later use in multiplicative iterations. In the third stage, multiplicative iterations were performed to calculate partial remainder and quotient until achieved required precision level using the Taylor series expansion algorithm.

VII. VARIABLE LATENCY CLASS

Till now, we have seen that division algorithms depend on retiring a fixed amount of quotient bits at the end of every iteration. In the case of the digit recurrence algorithms class, like restoring and non-restoring algorithm retiring single bit

in every iteration, the radix-based SRT algorithm has multiple possibilities, from one quotient bit to several quotient bits in one iteration, depending on the radix used to design the divider, e.g., radix-2 retires one quotient bit and radix-4 retires two quotient bits. High radix and look-up table class algorithms are also similar to the digit recurrence class. It shows the linear convergence towards its quotient detection, which suggests a fixed number of cycles until it reaches the quotient's final bit. In the case of the functional iterations class, the number of quotient bits retired in one iteration is greater in every iteration, but the number of cycles is fixed. As we have discussed, it is possible to reduce these dependencies and provide a solution with variable conversion time or latency time. Variable latency class algorithms are similar to the previous algorithms but with the possibility of a variable quotient bit retiring rate in different iterations or some iterations requiring less execution time, resulting in different conversion times in different sets of dividends and divisors.

The DEC Alpha 21164 is one of the best examples of variable latency class algorithm implementation and is based on the concepts of the simple normalizing non-restoring division algorithm. In DEC Alpha 2164 implementation, whenever the partial remainder is generating zeros or ones consecutively in the partial remainder, then similar weight quotient bits are also set to the sequence of 0's or 1's detected in the partial remainder [49]. It is found that the average number of quotient bits retired in one iteration varies from 2 to 3 depending on the stream of bits in the partial remainder. There are certain ways to provide a variable conversion time due to variable execution time in a particular iteration, given the fact that the execution of a particular combination of divisor and dividend in a particular iteration can be completed in a short time and normal execution time. It is possible to do so by saving very common bit combinations that result in early iterations and reusing that result in the next particular iteration. Ways to do so include

1. Self-timing
2. Result cache
3. Speculation of quotient digit

A. SELF-TIMING

In the self-timing technique, multiple stages are cascaded together with a self-timing partial remainder, suggesting no shift register is required to store the partial remainder in two cascaded stages. To match the timing of execution of the two cascaded stages, it has to self-time the partial remainder of the previous stage with the next stage, and thus the execution of the next stage with the generation of the partial remainder in the previous stage of the cascaded connection, providing overlapping of execution. It improves the latency by providing the average cycle time instead of the combined cycle time in cascaded stages. In [44], details have reported the implementation of a variable latency SRT algorithm-based divider. It uses five stages of cascaded radix-2 with the self-timing partial remainder, meaning no delay in transmitting the partial remainder to the next stage. Thus, in the next stage, execution

starts before the previous stage's quotient reaches the next stage. Hal SPARC V9 processor and Sparc64 are examples of practical implementation of the variable latency self-timing division algorithm.

B. RESULT CACHE

In typical division applications like an inversion of the matrix, square root, etc., it must perform repeated operations. In an inversion of the matrix, each and every term of the matrix is divided by the determinant, and in such cases, the possibility of repeating the same operands for operation is likely to be very high. Thus it is preferable to store the result of the operands in cache memory so that the next time the same operands perform a division; then this will just be copied from the cache. By recognizing such redundant behaviours or operations, or applications, it is possible to develop a variable latency divider. In [50], Richardson presents a result caching technique to implement along with the divider, resulting in a reduction of the conversion time. It allows a trade-off between the execution time and memory area. It provides a variable execution time on account of the large memory area. This caching results concept uses two stages: one is cache training, in which standard operation is executed depending on the reputability of the operands, and the result is stored in the cache memory. So, when a required operation is executed at that time, two events have started: one is the execution of operands, and the second is cache access. Suppose cache access results in the presence of a combination of operands. In that case, the result of that combination stored in the cache memory is transmitted and used further by terminating actual execution. In contrast, if there is a mismatch, then the operand's execution continues to get the result, and this result is stored in the cache memory. Thus, if the same operand combination arises in the next iterations, it does not need to be computed. Then it will take the result directly from cache memory, resulting in a reduction of conversion time. Using the cache to improve latency will affect the area efficiency; e.g., in radix-4, it needs to store 160 bits per cache entry.

C. SPECULATION OF QUOTIENT DIGIT

In [51], Cortadella mentioned implementing the SRT divider with variable latency, which detects a variable number of quotient bits in each iteration. This technique's main concept is to utilize fewer bits from the divisor and partial remainder than the normal radix-n divider utilizes. In this case, the accuracy of getting the correct quotient bit at the end of an iteration is uncertain. One extra iteration is required to rectify the incorrect conjecture in iteration due to too few bits for quotient bit selection. An increase in the number of iterations depends upon the degree of closeness when a correct quotient bit is detected in an iteration.

VIII. DIVISION HARDWARE ARCHITECTURE

To improve the electronic implementation efficiency of mathematical operators has two possibilities. The first one relates to improving algorithms that can be responsible for logical

data flow and conversion process in hardware. Simultaneously, the second one deals with improving hardware architectures, which are nothing but hardware interconnection and implementation for performing a mathematical computation. The first form of improvement is mostly considered because it takes less cost and time than a hardware change, which can cause 100 times costlier than soft changes like algorithm improvement. Even though we have to consider a better trade-off between soft changes and hardware changes for better improvement, because of the interdependency of software changes and hardware changes, better algorithms can be developed based on the best hardware, and the best hardware can be developed based on algorithmic needs. New algorithms are developing alongside old algorithms to efficiently perform the same operation, depending on new technological developments. The development of hardware and algorithm sometimes depends on the available situations required for a particular application. Depending on application requirements, old algorithms can be upgraded, or a new algorithm can be designed, or new hardware architecture can be developed. The timeline required to develop hardware is much longer and costlier than that of algorithm development. Thus it is preferred in most applications. At the beginning of the electronic era, things were analog, which took over decades to switch over digital, but algorithms are the same or modified more or less. Initially, after developing digital circuits and integrated circuits, hardware architecture classification falls into two broad areas; one is sequential or serial, and the other is parallel or concurrent hardware. Over a period, new hardware developed along with new and modified algorithms gives a different dimension to it, and we can have sequential-parallel, i.e., pipelined hardware architecture. It supports modification in the sequential algorithm, which could perform some operations parallel to improve efficiency. Sequential implementation requires less area and requires more time for conversion, whereas parallel architectures required a large area but very fast in conversion, and pipeline architecture is the best amalgamation of both.

In general-purpose applications, central processing units (CPU/processor) performs division with several iterations, even for a small number of bits. This problem goes critical, along with an increase in bit count [52]. Such problems are even more serious in the graphics processing unit (GPU) and Intel's many integrated core (MIC) architecture, which provides parallel architecture. The basic hierarchy of architectures goes from CPU, MIC to GPU. CPU works on the architecture consist of a single core, MIC works on multiple cores, whereas GPU works on several cores. Both GPU and MIC doesn't have any dedicated dividers unit or direct instruction to perform division [77]. Floating-point implementations performed on GPU and MIC with higher precision. The floating-point divider is implemented on the NVIDIA K20 GPU card with CUDA programming support that includes 30 different basic instructions and memory access. CPU's working frequency increased up to the 3GHz overtime period, and on the other hand, it increases the

power dissipation. An alternative to this is to use several CPU cores in parallel, which gives GPU or MIC exposure in general-purpose processing [54].

Each of the algorithm classes, which we have discussed in the previous sections, can be categorized into three categories based on the hardware architecture used for implementation. Serial hardware architecture consists of the sequential implementation of the algorithm's components required for algorithm implementation, basically used for general purpose. Processing in CPU or FPGAs; the best example of serial dividers is the simple restoring non-restoring digit recurrence algorithm. Simple architecture and ease of understanding are the main plus points of this technique, but it lacks latency, as the second iteration depends on the completion of the first iteration. The second technique used is parallel architecture. Multiple sets of hardware units are executed simultaneously to get the result in fewer iterations, basically used for graphical processing unit (GPU) or Intel's many integrated cores (MIC) processors. It is latency-efficient but lacks area efficiency. A third technique is a hybrid technique that essentially parallelizes the parallel execution to get the average area and latency efficiency. It performs certain operations in sequence and some in parallel, causing an average latency cycle instead of a combined latency cycle. The iterative divider structure shows the serial implementation of the division algorithm, whereas the array base implementation structure of the division algorithm represents a parallel and pipelined architecture for the implementation of the division algorithm, depending on the execution sequence. In the parallel architecture, the array starts execution of all array stages together, whereas in a pipelined architecture, the next stage's execution starts after a particular level of the previous stage is achieved.

A. SERIAL/SEQUENTIAL DIVIDER

Subtractive iteration based digit recurrence division algorithms is the best example of a serial divider, where iterations are interdependent to perform its operation. Hardware division by small integers occurs decimal to binary conversion, memory access. Online division, as Fig. 15, is also the best example of serial implementation of division. When are consider serial dividers, then there come two possibilities. When the input operands are provided sequentially like on-line dividers and others, the input operands are provided, but the iterative conversion process works serially to converge quotient linearly [17], [69], [84], [111]. The long division, which resembles the theoretical paper-n-pencil algorithm, is also a sequential subtractive algorithm. The basic idea of a radix-n algorithm also performs sequential iterations based on radix number n [69]. Many efforts have been made to make the sequential process faster, and the most efficient and successfully implemented method is the SRT algorithm. Many processors like Pentium has implemented this division algorithm. As discussed in the previous section, this method's major drawback is it needs a careful design of quotient digit selection logic in the overlapped region. General-purpose processing applications demand improvement in simple and

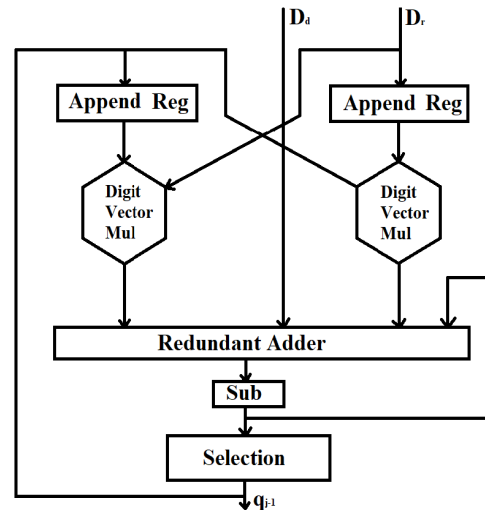


FIGURE 15. Operational block diagram of the online division algorithm.

eased algorithms. One of them is data-dependent dividers, which work to avoid redundant operations and perform only shift over zeros operation used in the SRT algorithm to normalize the remainder. They can introduce higher throughput than radix-2 dividers, so it is useful to use them in small architectures [69], but the conversion speed is currently not implemented in other architectures. SRT algorithm can be implemented on different architectures [17]. As we have discussed, the division's serial operation, in on-line division prescaled operands are provided serially. Thus the quotient generation rate depends on the rate at which input operands are provided [84]. The output generates upon completion of δ clock cycles after the first digit from the first operand is supplied to dividers. This on-line delay conversion time varies from conversion to conversion depending on the size of the input operand, number system, and radix used. The on-line division [84] is represented as

$$S[j] = rS(j-1) + D_{dj+\delta-1}r^{-\delta+1} - rq_{j-1}D_r(j-1) - D_{rj+\delta-1}Q(j-1)r^{-\delta+1} \quad (54)$$

where,

$$D_d[j] = \sum_{i=0}^{j+\delta-1} D_{di}r^{-1} \quad (55)$$

$$D_r[j] = \sum_{i=0}^{j+\delta-1} D_{ri}r^{-1} \quad (56)$$

$$S[j] = \sum_{i=0}^j S_i r^{-1} \quad (57)$$

$$Q[j] = \sum_{i=0}^j q_i r^{-1} \quad (58)$$

$$S[j] = r^j (D_d[j] - D_r[j]Q[j]) \quad (59)$$

It indicates that S is a scalable residual value defined as (59). Quotient digit is selected based upon quotient digit selection logic similar to that used in SRT algorithms. Higher radix in on-line serial dividers yields fewer iterations and potentially better performance on account of a large area and longer cycle time.

B. PIPELINED DIVIDER

Pipelined architecture is one of the distinctive outcomes of performance enhance efforts. As dividers applications are increasing, the need for high performance (area, time, power) dividers is increasing. Clock cycles required for integer division is unexpectedly long and uncertain [6]. A pipeline architecture is one of the keys to improving overall computational performance. This architecture allows performing several instructions of the computation process simultaneously to achieve some degree of parallelism. Pipeline architecture provides parallel processing by performing the instruction-level overlapping of a computational process [6], [11], [17], [61], [84]. The execution of pipelined architecture is very similar to that of the production-line workflow. Every working point worked on a specific task and passed on the task to the next level. Likewise, when the task is in the second level, the first level can start a new task; thus, it looks like a parallel working. Pipeline work structure can be achieved by designing a computational logic that will provide functional overlap in the execution stage and by arranging pipelined hardware like a fully pipelined array structure [6], [11]. In short maximum serial computational algorithms can be performed using pipeline architecture.

SRT division algorithm is the best example of this implementation. Functional iterative division algorithms are based on multiplicative iteration, which almost required no extra hardware to work on pipelined architecture whereas, in the case of digit recurrence algorithms like SRT algorithm, which works on subtractive iteration. The subtractive iteration algorithm requires separate hardware at each stage; thus, the SRT algorithm needs to use separate addition, subtraction, and shifting in each computational cycle, causing increased complexity and the size of the quotient digit selection logic look-up table [6], [11], [61]. As we discussed, the implementation of on-line serial dividers in the previous section requires a different clock cycle depending on the operand digit count. To use pipelined architecture in on-line dividers, we need to have a two-stage pipelined implementation of on-line dividers, as shown in Fig. 16. The first stage will compute the remainder's partial value and append the new divisor to the vector value of the divisor. In the second remainder, and the next quotient digit is calculated. Working of 2 stage pipelined on-line dividers can be expressed as (60)

$$S[j] = rS[j-1] + D_{dj+\delta-1}r^{-\delta+1} - rq_{j-1}D_r[j] - D_{j+\delta-1}Q[j-2]r^{-\delta+1} \quad (60)$$

The critical path is reduced in the second stage of dividers as compared to normal implementation. Block-level implementation of two-stage pipelined on-line dividers is shown in Fig. 16.

C. PARALLEL DIVIDER

As we previously stated, the division requires larger latency as compared to other operators. Even though it rarely occurs in general-purpose computing, it is the most necessary operator

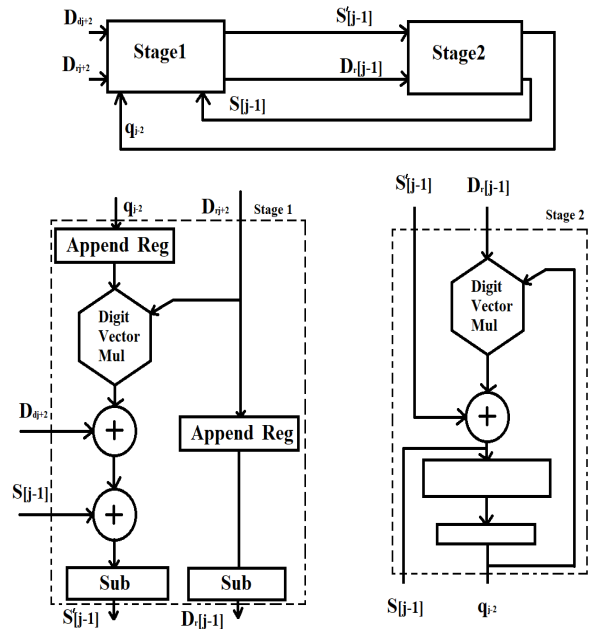


FIGURE 16. Block-level implementation of 2-stage pipelined online divider.

in applications like vector calculation, complex number calculation, artificial intelligence, graphics processing unit, etc. [8]. Sequential / serial implementation could be tricky to achieve high speed and accuracy. Thus these applications require a high degree of parallelism in architecture [8]. The basic idea of parallelism indicates simultaneous working or computing of the same operation. In a basic way, there are possibly two ways to achieve this parallelism. One is to optimize implemented hardware architecture, and the second is to optimize soft processes or algorithms [8], [52], [55], [112], [113]. Considering the optimization of hardware means to redesign the hardware would cost most and time consuming, which means to upgrade integrated circuit chips used in processors. On the other hand, optimizing soft process means upgrading computational algorithms in software to make optimal use of existing hardware [52], [55]. Graphics Processing Unit (GPU) and Intel's Many Integrated Cores (MIC) hardware are the best examples of parallel architecture used for computation. Intel's MIC architecture consists of a few cores parallel with no direct hardware to compute division. Whereas, in the case of GPUs, it considers several cores in parallel [8], [52], [55], [112], [113]. Such parallel architecture is best suited for the systems that work on the bits n pieces of data, i.e., data packets like digital signal processing in which parallel architecture gives multi-thread computation possibilities. GPU and MIC work on a large number of the multidimensional array data structure for numerical computing techniques. It generates an opportunity to explore the use of single instruction multiple data parallelism (SIMD) techniques, which exhibits property convergence through iteration of several stages to achieve a certain condition.

1) SMALLER DIVIDEND DIVISION ALGORITHM

It is the simplest algorithm in terms of complexity when we come to parallel computing. When we are implying high radix dividers, it is one option to use multiple small radix dividers in the pipelined or cascaded form to achieve the required solution. As the use of small radix dividers for implementing high radix yields the desired solution but on the other hand, it increases the complexity, area, and performance ratio to cost factor [8]. The basic phenomenon behind this algorithm explained in [8], [31] is to consider division as a fraction. Thus by applying properties of fractions, it can reduce the complexity associated with the parallel division. Consider two unsigned numbers for dividend and divisor. Consider dividend bit count as 4n and divisor bit count as n. We can represent dividends in terms of partitions based on associated weights. Then we can represent the dividend as the addition of number partition as (61-64).

$$N_1 = \sum_{i=0}^{2n-1} x_{2n+i} 2^{2n+i} \tag{61}$$

$$N_2 = \sum_{i=0}^{2n-1} x_i 2^i \tag{62}$$

$$D_d = N_1 + N_2 \tag{63}$$

$$D_d / D_r = (N_1 + N_2) / D_r = N_1 / D_r + N_2 / D_r \tag{64}$$

Fig. 17 shows the basic implementation of the algorithm. Thus by calculating the total of fractions, we can derive an actual solution for the division. The algorithm consists of three stages as

- In the first stage, a preprocessing stage here performs dividend partitioning depending on the radix.
- In the second stage, the iteration stage performs iterations to compute a division of partitions and generate partial quotient and partial remainder.
- In the third stage, the combining stage operates to obtain the final quotient and remainder.

For a better understanding of the algorithm, steps involved in radix-2 division using a smaller dividend algorithm is gives as

- In stage one, based on radix-2, make two partitions of 4n bit size dividend.
- In stage two, perform iterations in N1 and N2, partitions of 2n bits, forming q1,q2, r1,r2 as partial quotient and partial remainder.
- In stage 3, combine the partial remainder and partial quotient to receive the final quotient and remainder.

The simplicity of conversion logic is an advantage and reduces the latency on account of increased hardware requirements as it requires separate dividers equals to the number of partitions performed with a dividend. It also exhibits some limitations like the need for a higher dividend than the divisor, synchronization between parallel units, a special focus on recombining logic for partial quotient, and the partial remainder to generate the final quotient and remainder. This algorithm's major benefit is that it can use any existing method to compute partial division in the iteration stage. It gives a better tread between area and time to choose the desired combination as per the application needs.

2) JEBELEAN EXACT DIVISION ALGORITHM

We perform complete division on long integer operands in digital computations even after knowing that the remainder will be zero. It causes unnecessary computation time. In such cases, Tudor Jebelean proposed an algorithm in 1992 to use the advantage of division being exact. It proposes to work starting from the least significant digit of operand [114]. Implementation of this algorithm works remarkably well only when radix is prime or power of 2. The algorithm uses the least significant digits operand first to generate the least significant bits of the quotient, making it significant for pipeline or parallel implementation. Pompeiu introduced the basic idea of the algorithm in 1959 that uses only the least significant digit of operands to find the least significant digit of the quotient as operands and quotient are represented multi-precision positive integers expressed as radix -n. As the division is considered as exact thus, it can show $D_d = d * Q$. We can Express the algorithm [112]-[114] as

$$D_d = \sum_{i=1}^m D_{di} n^{i-1} \tag{65}$$

$$Q = \sum_{i=1}^m q_i n^{i-1} \tag{66}$$

D_{di} and q_i are the least significant digits of dividend and quotient stored in the least significant digit first format. Thus after k^{th} iterations, we get

$$D_{dk} = \sum_{i=1}^k D_{di} n^{i-1} \tag{67}$$

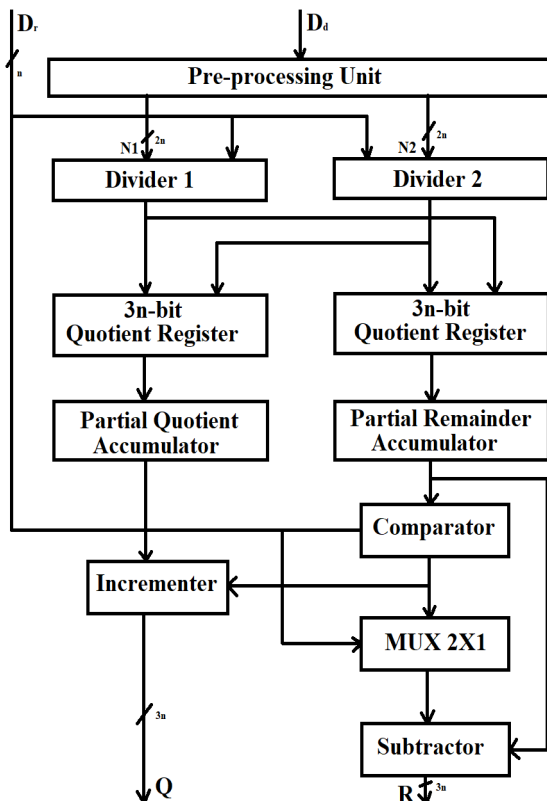


FIGURE 17. Block-level implementation of small dividend division algorithm.

$$D_{dupk} = \sum_{i=k+1}^m D_{di}n^{i-1-k} \tag{68}$$

$$Q_k = \sum_{i=1}^k q_i n^{i-1} \tag{69}$$

$$D_d = D_{dupk}n^k + D_k \tag{70}$$

where D_{dk} , D_{dupk} , Q_k and b_k as a state of the Jebelean algorithm D_{dk} is a multi-precision number formed by $D_{d1}, D_{d2}, D_{d3}, \dots, D_{dk}$. D_{dupk} is the upper part of the dividend $D_{dk+1}, D_{dk+2}, \dots, D_{dm}$. The actual dividend can be represented as the (70), and the quotient is the multiple-precision quotient after the k^{th} iteration. To implement the Jebelean algorithm parallelly, it needs to borrow b_k calculation in parallel, which is quite challenging. Parallel computation of borrow b_k is expressed as

$$b_k = \left(-n^{-k} D_{dk} \right)_{mod d} \tag{71}$$

$$b_k = \left(-n^{-k}_{mod d} D_{dk} \right)_{mod d} \tag{72}$$

Assuming we have n processors, we can assign k^{th} processors to perform a calculation to find out b_k value and then q_k value. Each processor will compute $D_{dk}n^{k-1}$, then run a parallel prefix sum and then multiply by n^{-k} . Execution is performed by modulo d . Takahashi’s algorithm uses a slightly different approach to derive left to right Jebelean algorithm. In Takahashi, the remainder is executed sequentially, and if the remainder could get parallelly, then the final quotient could get parallelized. Recurrence for the remainder [112], [113] is given as

$$r_k = (nr_{k+1} + D_{dk})_{mod d} \tag{73}$$

$$r_k = \left(D_{dk} + nD_{dk+1} + n^2D_{dk+2} + \dots + n^{2n-k}D_{dkm} \right)_{mod d} \tag{74}$$

Takahashi uses a parallel cyclic reduction method to solve the remainder recurrence. The general form of i^{th} iteration

$$r_k^{(i)} = \left(r_k^{(i-1)} + n^{d(i)} r_{k+d(i)}^{(i-1)} \right)_{mod d} \tag{75}$$

where look ahead distance is $d(i)$, it gets doubled at each step; thus, $d(i) = 2^i$. It is also called a short division or exact division.

IX. IMPLEMENTATION STATISTICS

A variety of applications has implemented many division algorithms. One has to select an appropriate algorithm that can cater to the cost, area, time, and complexity requirements of applications and technology to manufacture. This section presents a study of different division algorithm implementations, which gives a broad insight into the different implementations. Upon this, one can understand the necessity to choose a proper trade-off between time, cost, area, and complexity while selecting the proper algorithm that can be suitable for fulfilling an application’s requirements. Considering the simplicity and vast variety of implementation possibilities had before and would come in future digit recurrence algorithm is mostly the choice of interest for many applications, but it is

very experimental to visualize the different implementation aspects of various algorithm which could lead towards new ideas to improve some old implementations or to develop a new one.

TABLE 3. Summary of Handel-C implementation comparison.

	LUT's	Frequency (MHz)	
		From	To
Handel - C	747	7,21	10,965
Restoring	115	13,716	20,345
Non restoring	144	24,175	40,073
Non restoring with pipeline	66	37,806	63,558

Restoring and non-restoring algorithms are very broad concepts. The restoring algorithm resembles the actual long division algorithm or, never the less with theoretical paper n pencil algorithm, and a non-restoring algorithm is similar to restoring except restoring stage. These are the basic algorithms of the digit recurrence class of dividers. Many algorithms come later, which are fully or partially derived based upon non-restoring algorithms ideology. Many researchers [4], [5], [15], [18] have explained the complexity, timing, area, and other features related to the implementations of basic restoring and non-restoring algorithms. D. G. Bailey [4] presented an article about the statistical implementation data for restoring and non-restoring algorithm in 2006. In this article, he presented a comparative analysis of FPGA and Handel-C implementation of restoring and non-restoring algorithm. Algorithms were implemented on RC-100, RC-300 development boards produced by Celoxica using Xilinx’s Spartan-II and Virtex-II FPGA. Restoring and non-restoring dividers were built as macro expressions with Handel-C language and compiled to generate EDIF file within Celoxica DK4.0 environment further EDIF file is mapped with respective FPGA of RC-100 and RC-300 board using Xilinx ISE version 6.1.03 [4]. Handel-C is very similar to that of the C programming language with the additional benefit of inherent parallelism property [115], [116]. A statistical comparison is presented between algorithms implemented as macro expressions with Handel-C built-in integer divider. It is to be considered that, for comparison, only restoring and no restoring algorithms based on basic equations expressed in the earlier section are used without implementing the radix SRT algorithm. The comparison presented in table 3 concludes that Handel-C built-in divider is the slowest as it can work on frequencies near 10 MHz. The chip area required in FPGA is approximately more than double the chip area required by designed algorithms. In Handel-C implementations, it indicates that the use of subtraction for performing a comparison and reusing it as an input to a multiplexer and using separate LUTs for addition and multiplexing

requires extra hardware limiting speed improvement. Implementation of the basic idea of restoring and non restoring division algorithm could implement in a sufficiently low chip area, but the maximum working frequency is low. The number of LUT's may vary based on considering HDL languages to implement the above algorithms. Many non-restoring algorithms were designed and implemented, but the SRT algorithm is the most implemented. The basic SRT algorithm was implemented in [7], [11], [17], [51], [57], [63], [64], [67], [69], [71], [32], [74], [76], [78], [81] for different applications utilizing different aspects of algorithm.

In [4], E. Matthews, A. Lu, Z. Fang, and L. Shannon discussed integer divider designs for FPGA based soft-processors ascendancy over patronage of adaptation of variable latency execution unit in their instruction pipeline. Implementation efforts were focused on the Quick-Div divider, which shows data-dependency and variable-latency in integer division. It integrated into the FPGA-based Taiga RISC-V pipelined soft-processor. Comprehensive results compared with fixed latency radix-2/4/8/16 dividers. It has been mentioned that dividers also are classified into two types. One as fractional dividers (floating point) and the second as integer dividers. Integer dividers also have several applications in today's digital world, from simple pseudorandom number generators to complex applications like image processing, signal processing, etc. [7], over decades. It has been followed to use floating-point/fixed-point divider with a sufficient degree of numerical precision for working on integer division, sometimes on FPGAs, but hardware constraints are always there indicating the use of floating-point/fixed-point divider for integer division cause wasting of resources. It points out that a 64-bit floating-point/fixed-point divider requires almost ten times more resources than a radix-2 divider [7], [26]. FPGA soft-core processor, Micro Blaze [27], NIOS II [28], and the LEON3 processor [29] implemented fixed-latency radix- 2 dividers with 32 cycles of latency for performing division operation. In general basic arithmetic operations required two to three cycles, whereas radix –2 requires 32 cycles, making it comparatively slower with respect to others. Experimental implementations have been performed over the Xilinx Virtex UltraScale+ VCU118 board (XCVCU9P-L2FLGA2104E) using Vivado 2018.3 synthesis. In this article, they have given a comparison of different radix – n and Quick-Div dividers. Table 4 compares different dividers based on working frequency, LUTs, FFs, etc., when implemented stand-alone. With ascendancy over the variable latency execution unit's patronage in the Taiga soft-processor instruction pipeline, all dividers are realized with the RISC-V Taiga soft-processor. A comparative statistic is derived between the implementation of data dependant variable-latency Quick-Div dividers and fixed latency radix-n (n = 2, 4, 8, 16) dividers with and without the RISC-V soft-processor Taiga. Taiga is RISC-V open-source soft processor. Quick-Div dividers are unsigned processes, so that sign conversion before and after completing conversion is required depending on the instruction operands

TABLE 4. Summary of comparison between stand alone implementation based on LUTs, cycle and frequency.

Name	Cycles		LUTs	FFs	Frequency (MHz)
	Min	Max			
Radix-2	32	32	100	100	900
Radix-4	16	16	250	150	725
Radix-8	11	11	500	75	475
Radix-16	8	8	700	200	320
Quick-Div Initial	1	32	300	100	300
Quick-Div count leading zeros	2	33	350	170	400
Quick-Div CLZ-2BIT worst case optimization	2	33	450	170	300

TABLE 5. Summary of comparison between taiga soft processor implementation based on LUTs, cycle and frequency.

Name	LUTs	FFs	Frequency (MHz)
Radix-2	1500	1100	375
Radix-4	1520	1200	350
Radix-8	1990	1000	350
Radix-16	2100	1200	300
Quick-Div Initial	1600	1000	350
Quick-Div count leading zeros	1600	1150	375
Quick-Div CLZ-2BIT worst case optimization	1700	1100	300

and type. Due to this, Quick-Div requires additional 3 cycles for sign conversion, as mentioned in Table 5.

In [11], N. Sorokin discussed the implementation of fixed-point dividers based on different algorithms on Xilinx FPGA's common platform. Different divider modules have been compared with Xilinx's 32-bit IP core pipelined divider. It indicates that the non-restoring algorithm based fixed-point divider module is particularly faster than 32-bit Xilinx's IP core pipelined divider. In this article, it is pointed out that, in practical division operation results are more of approximated values than exact values in digital operations. These approximated values can make some trouble in more critical applications, like biomedical applications, sensors signal processing, coordinate computation for an item, etc. [11]. As we have discussed earlier, even for integer division, we have to use the fractional divider, which includes a fixed point or floating-point divider; thus, floating-point implementation is critical and complex, making it sometimes impracticable. Out of many theoretical concepts, one practicable solution was provided by Xilinx's IP core pipelined divider. Still, 32-bit input operands cause to produce 32-bit remainders in many cases, which is impossible to implement in applications

TABLE 6. Comparison based on conversion time of Xilinx IP core and other divider module.

Parameters	Time of conversion (ns)		
	8	16	32
Xilinx IP core	211	253	350
SRT	525	635	854
Non Restoring	165	198	265
Restoring	324	405	597

TABLE 7. Properties of 32-bit IP core pipelined divider.

Number of bits	Properties			
	Slices / LUTs	FFs	Look-up tables	Frequency (MHz)
8	2247	4020	1400	204,3
16	2742	4904	1680	201,6
32	3843	6864	2240	193,1

TABLE 8. Comparison with restoring and non-restoring dividers.

Number of bits	Frequency (MHz)		
	Xilinx IP core	Non Restoring	Restoring
8	204,3	250	130
16	201,6	248	115
32	193,1	245	100

where high precision in calculations is required. Another implementation problem-focused in this article is about the chip area requirements of this solution. The fixed-point algorithm follows the basic principles like simple paper n pencil division algorithm. A fixed bit length quotient is generated in every iteration of fixed-point divider like digit recurrence type of dividers. A major focus was given on improving addition and multiplication operations, as speeding up addition operations reduces computational time in the actual division process. Replacement of divisor by its inverse value can allow multiplying by anti-divider to obtain division result. Speeding up dividers has been achieved by developing fast adders, carry look-ahead adders, matrix or array type adders, etc. Xilinx’s IP core divider has certain properties:

- It is available in drop-in modules for all Virtex, Virtex-II, Virtex-II Pro, Virtex-4, Spartan-3, etc.
- The dividend can be up to 32 bits.
- Full pipelined architecture.

Table 6 - Table 9 shows the implementation properties and comparison of the 32-bit IP core pipelined divider by

TABLE 9. Other comparison with restoring and non-restoring dividers.

Parameters	Slices / LUTs			FFs		
	8	16	32	8	16	32
Xilinx IP core	2247	2742	3843	4020	4904	6864
Non Restoring	126	142	159	178	180	186
Restoring	160	180	200	178	195	210

Xilinx. The comparison is made with 8-bit, 16-bit, and 32-bit operands, which generate similar bit remainders. Even though the IP core divider gives an improved calculation speed, it still requires a large area and storage to store look-up tables for its performance. Fixed-point division core from Xilinx results in difficulty implementing it universally in every application due to some restrictions imposed by its implementation requirements as

- A large area occupied by the design and the limited widths of the operands, and the division’s fixed-point result.
- If there is a need to increase the result’s precision, one must find some ways to apply different operands’ scaling techniques.

In [16], Md. F. Kasim, T. Adiono, Md. Fahreza and Md. F. Zakiy discussed a divider block with pre-computed values stored in read-only memory in terms of a look-up table. This divider working is similar to that of dividers based on functional iteration type of algorithms like Goldschmidt’s algorithm and Newton’s method [16]. Thus, the result of this divider is also approximate value, unlike iterative subtraction class-based dividers. Table 10 gives a comparative analysis of the pre-computed divider concerning other implementations. In this case, they consider the same bit size numerator (N) and denominators (D), assuming $N, D > 0$ and $N < D < 1$.

Steps of algorithms are given as

- Scaling N and D so that D has a value between 0.5 and 1.
- After scaling the denominator, find the value of $x = 1/D$ from pre-computed values stored in the look-up table.
- Then multiply the value of x with the numerator, which is similar to optimizing the division algorithm to speedup division operation [11].
- Suppose we take p most significant bits out of n bits of D and reserve 2p items of pre-computed values, which cause an error. Thus $Y = N/(D_K + D_L)$. D_K is the p most significant bits of D and D_L remaining, i.e., n-p bits.
- Perform Taylor expansion to the above point, and we will get $Y = N/D_K (1 - D_L/D_K + D_L^2/D_K^2 - D_L^3/D_K^3 + \dots)$.
- 2p memory is utilized to save pre-computed values if we consider p bits, and the possible error can be 2-p. It makes it critical to select the value of p to optimize performance and control the memory utilization

in storing look-up tables and maintaining error conditions within acceptable limits. The maximum error of 0.187595 can occur when $p = 1$, which gets improved with an increasing number of p 's, so the maximum error of 0.000051 will come for $p = 15$ at the same time memory requirement will increase from 21 to 215.

In [8], K. Tatas, D. J. Soudris, D. Siomos, M. Dasygenis, and A. Thanailakis discussed the different concept of partitioning the main dividend in segments so that it represents an actual division of the numerator by denominator as a series of smaller division. By considering the weight of the dividend bits, all intermediate operations are performed. This concept of a series of divisions showcases a smaller dividend division algorithm, where we have to perform shifting, partial division, and accumulation operations. Any existing division algorithms can be utilized for the partial division process; the best-suited option must be selected depending on the trade-off between cost and area. Implementation of this algorithm is possible in both series and parallel ways [8]. A higher radix system is critical and difficult to implement, and its performance is not very high. In digital signal processing, field data is available in a series of bursts like packets, making throughput requirements more critical over latency. The concept is to divide a large numerator into multiple smaller parts, i.e., partitioning into fixed numbers with its associated weight, then divide this small numerator by a denominator. At last, to add all small divisions to give a result.

$$\frac{N}{D} = \frac{N1}{D} + \frac{N2}{D} + \frac{N3}{D} + \frac{N4}{D} + \dots \quad (76)$$

The partitioned numerator's partial division process can be performed either serially or parallel due to the tread between cost and time. This algorithm is implemented with a length of $N = 32$ -bit dividend and parallel array divider, sequential divider with two partitions, or parallel divider with two partitions in the partial division stage. Its respective implementation required 4316, 2136, and 3050 slices on Xilinx Virtex-E 1000. From the above data, it is clear that sequential implementation of this algorithm is more area efficient and moderate in time delay. If any corrective stage is required in sequential dividers, it will degrade the efficiency of serial dividers. In contrast, parallel implementation produces a slight reduction in delay but not a sufficient decrease in area and latency. Array Implementation of this algorithm is not at all efficient as it increases chip area four times on doubling word length.

In [30], J. Liu, M. Chang, and C-K. Cheng discussed an algorithm that utilizes prescaling, series expansion, and Taylor series expansion together; hence it is sometimes called a PST algorithm. At the starting, both operands are prescaled up to it reached to the suitable starting level. Operand prescaling is performed based on the scaling factor E_0 , which is stored in the look-up table. In the second stage of the PST algorithm, series expansion is applied on scaled operands to obtain an accurate anti-divisor approximation. To calculate the partial quotient and the next remainder in the iteration

TABLE 10. Comparison of 32 bit pre computed divider along with other dividers.

Parameters	TLEs / LUTs	Latency (uS)	RMS error
Pre computed divider	647	$3,22 \times 10^{-2}$	$4,37 \times 10^{-4}$
Goldschmidt's algorithm	816	$3,82 \times 10^{-2}$	
Non-restoring radix 2	676	$5,75 \times 10^{-2}$	
Divider from Quartus Mega functions (32 bits)	1146	$15,3 \times 10^{-2}$	

TABLE 11. Implementation statistics of PST divider.

Parameters	Frequency (MHz)	LUTs	Memory	DSP
IP core from MegaWizard	50,16	1203	84	0
PST (DSP)	72,8	213	768	28
PST (non DSP)	73,2	1437	768	0

stage, it utilizes 0-order Taylor series expansion. Iterations have to continue until getting the quotient with a required precision range of error. Three Taylor expansion iterations and a look-up table are needed to finish one operation. As per the performance comparison with the IP core and DSP and non-DSP structure of this algorithm shown in Table 11, the divider shows significant delay and doesn't save sufficient area than Xilinx IP core and some other divider design. PST divider is FPGA feasible, and new placing routing and packaging techniques may generate an improved version of the PST divider.

In [81], A. Vazquez, E. Antelo, and P. Montuschi presented the SRT algorithm base radix-10 architecture to work as a floating-point divider. It works on the basics of the SRT algorithm like sign digit (SD) redundant digit range for quotient and digit selection logic design on constant comparison of carry-save estimation of the partial remainder. These showcase the alternate use of the BCD number range for representing decimal operands instead of regular weighted binary arrangement. Basic SRT implementations show the generation of odd multiples of divisors in the radix- 2^k high radix system, which could degrade the implementation. It could be resolved using simple overlapping of two recurrences of low radix-n systems, but in this system implementation, it shows that odd multiples of divisor can be generated by simply using decimal carry propagated adders and resue further. To represent operands in signed digit range, it uses 10's complement representation of bit length 4. Table 12 gives details about implementing the BCD system for the floating-point divider. A delay is represented as a delay term of multiple of an

TABLE 12. Implementation statistics of BCD floating-point divider.

Parameters	Slices / LUTs	
	Area (No. of NAND gates)	Delay (FO4)
Selection logic	3200	22.3
Multiple generator	2000	18.4
Adder	2600	21.8
Mux/Latch	2700	3.0
Total	10500	25.3

inverter with a fanout of 4. The rough estimation of hardware size and complexity is given in multiples of the minimum equivalent area of a two-input NAND gate.

Many different applications are possible for radix-n base non-restoring digit recurrence algorithm. In [86], [87], M. D. Ercegovac and R. McIlhenny present the implementation of radix-10 with limited precision primitives, which uses modules of 1 to 3 or 1 to 4 decimal digits. The proposed method is based on the use of limited precision multipliers, adders, and look-up tables. Minor changes have been suggested at the initial stages to work with limited precision, such as using the Fourier series to achieve the desired recurrence for limited precision primitives. It produces one quotient digit per iteration using shifted short partial remainder and short anti-divisor or reciprocal. Implementation is performed using Xilinx's d10.1 and 12.1 design suit tool and mapped to Xilinx's vertex-5 and vertex-6 FPGA. Total delay can occur due to a short reciprocal look-up table (T_{rec}), selection function (SEL), a digit by digit multiplier along with compensation factor (C-net), auxiliary residue (V-net), Next residue (W-net), and on-the-fly conversion for signed digit conversion to conventional decimal representation. The implementation of 1 to 3 decimal digit short reciprocal for significant size $n = 7$ and 14 respectively requires 782 LUTs, 105 ns approx delay, and 1263 LUTs, 197 ns approx delay. Implementing 1 to 4 decimal digit short reciprocal for significant size $n = 7$ and 14 respectively requires 1384 LUTs, 102 ns approx delay, and 2047 LUTs, 204 ns approx delay. The main lacking of routing delay indicated in implementation is very high.

In [90], M. Baesler, S.O. Voigt, and T. Teufel presented the implementation data for shift and subtract algorithm, digit recurrence algorithm with signed redundant quotient, and carry-save representation. The second representation uses ROM to calculate the quotient digit, whereas, in the third representation, the quotient is derived from digit decomposition without ROM. Type 1 uses a simple shift and subtract algorithm for the fixed-point divider, which indicated unsigned and non-redundant quotient digit calculation. Type 2 uses signed digit calculation for quotient digit with redundancy factor 8/9 with operand scaling to get divisor in range in

between 0.4 to 1.0. Type 3 uses divider scaling to calculate quotient digit, where the divisor is prescaled in between 0.4 to 0.8 with redundancy factor 8/9. For normalized decimal fixed point divider, type 1 divider requires 3868 LUTs and FF in combine total latency of 154 ns and maximum working frequency of 123 MHz. Type 2 requires 2210 LUTs and FF in total and can work up to 118 MHz max frequency and provide a latency of 162 ns. Type 3 requires 2203 LUTs and FF in total and can work up to 88 MHz max frequency and provide a latency of 230 ns.

In [91], M. D. Ercegovac, and J-M. Muller proposed a digit-recurrence algorithm for real and complex number division. The concept presented in this article indicates the use of a variable radix divider as a key element along with prescaled operands by using sufficiently low radix. It elaborates the method of using a low radix conversion to high radix during iterations post initial estimation in the first iteration. Implementation parameters are given in comparison with the area and delay of the full adder. Thus, the total delay is counted as the overall delay that occurred in all building blocks like registers, adders, multipliers selection logic, multiplexers, etc. It estimates the proposed scheme's area requirements up to radix-256 with internal precision of 64 bits in total 1750 to 1880 times the full adder area. In [61], B. Mehta, J. Talukdar, and S. Gajjar present high-speed SRT dividers based on a highly parallel pipelined structure with fuzzy logic quotient digit selection proposed a high level of parallel performance of execution steps. It represents design implementation based on parallel SRT radix-4 module algorithm, which initiates prediction based on dividend and later correction made by fuzzy logic to reduce Q selection logic look-up table size. For 64-bit double-precision floating-point number required 1879 LUTs, 283 Registers with a lowest critical conversion time came out to be 210 ns. In [110], B. Pasca presented a piece-wise polynomial approximation and Newton-Rapson algorithm for the division for DSP supportive families of FPGA from Altera. As a basic problem of the Newton-Rapson algorithm, it contains some rounding errors. Thus to overcome this problem associated with the Newton-Rapson method, it proposes using highly tuned piece-wise polynomial approximation, which provides faithful rounded implementation with one extra bit of precision. This method is similar to dewpoint rounding. Synthesis results for floating-point implementation of the proposed method required 274-426 ALUT, 291-408 Registers, 3-4 DSPs for a polynomial approximation of $d = 2$. For a polynomial approximation of $d = 4$ required 1113 ALUT, 1825 Registers, and 9 DSPs. For a polynomial approximation, $d = 2, 4$ and Newton-Rapson required 887-947 ALUT, 823-1296 Register, and 9 DSPs.

In [52], K. Huang, and Y. Chen proposed a fast approximation algorithm to estimate the floating-point numbers in IEEE 754 format. It consists of two parts one is the prediction stage, and another is the iteration stage. The floating-point division is normally required several cycles to complete its execution on CPU base architectures where serial or pipelined

implementation is used, whereas in the case of GPU and MIC architectures where they don't have any direct instruction to perform division, this situation is more critical and required more cycles to complete the division operation. In this proposed method, improvement of time performance is the prior focus to achieve than the area requirements. In the iteration stage, the Goldschmidt algorithm with the binomial theorem is used as it indicates a fast convergence rate and ease in implementation using fused multiplication and adder (FMA) construction on MIC and GPUs. The iteration step improves the precision of result approximation based on an initial estimation of anti-divisor or reciprocal. In this implementation, lower degree polynomial approximation cannot reach the required level of precision, and a higher degree of polynomial approximation increases the complexity of calculation. A technique is applied to overcome the problem by considering floating-point numbers as a fixed point integer and perform simple integer subtraction to generate the required accuracy in estimating anti-divisor or reciprocal. Prediction stage maximum allowed error has no value more than 0.06.

Implementation results were generated on NVIDIA's K20GPU having 2469 cores with CUDA 5.0 compiler and 0.71GHz working frequency. The implementation was compared with Intel's Xeon Phi 5100 Series MIC having 60 cores with Intel composer XE 2013.2.146. It indicates that the initial prediction stage error up 0.0508 to 0.0614. It took 117.7 GFlops on MIC K20, while built-in implementation of CUDA takes only 46.6 GFlops, and on MIC, it requires 3736 GFlops. In [113], N. Emmart, and C. Weems presented a multi-precision integer division algorithm by single-precision value using GPU. The proposed algorithm is based on the parallel version of Jebelean's exact division algorithm with left-to-right borrow chain computation. Further improvement in precision is achieved by implementing Takahashi's cyclic reduction technique. Results show that the proposed parallel algorithm worked 20% slow in a 1024 bit size of dividend but shows 40% faster performance for 2048 bit size of dividend than Takahashi's algorithm.

X. COMPARISON

Even though the division operation looks simple, it is very difficult to implement due to strict conversion rules, and an efficient system needs to implement an efficient divider. Many algorithms were discussed in the previous sections, stating different logical concepts of achieving the division operation. It is very complex to differentiate all the implementable algorithms into independent classes, but there are broadly four. The first uses digit recurrence, which is also an iterative type of division. The best examples of this class are the restoring, non-restoring, and radix-n based SRT algorithms, in which a specific number of quotient bits are discovered in each iteration. The restoring and non-restoring algorithms work on iterative subtraction, whereas the radix-n based SRT algorithm works on predicting the quotient bit depending on a few MSB bits of the divisor and partial remainder followed by subtraction. The second functional iteration is

an approximation type of division. The best examples of this class are the Newton-Raphson algorithm and the series expansion algorithm. In the third, the look-up table stores a logic of quotient bit selection or pre-computed values that can be used in each iteration to detect the quotient bits in that particular iteration. This can be used along with digit recurrence or functional iterative algorithms. The fourth class is variable latency, which has a basic requirement of variable conversion time. One can design a division algorithm based on the nature of any one of these classes or an interdependent nature for better efficiency in implementation. The area, latency, and criticality of the quotient bit selection logic are the main trade-off points.

Given the continued industrial growth and technological improvement, there is a demand for achieving an efficient trade-off between the area, latency time, and criticality of the conversion logic. Operand pre-scaling and a high degree of redundant sets in quotient bits are two techniques commonly used for reducing the latency time. In the case of a radix-4 divider operand, pre-scaling can reduce the number of bits selected from the partial remainder and divisor for the quotient bit selection logic, which can improve the conversion speed by reducing the latency time. Simple staging (cascading), overlapping execution like overlapping quotient selection, or overlapping partial remainder computation in the execution of the SRT algorithm are also methods used to reduce the latency time on account of the extra area due to the extra hardware required for the implementation of performance-improving techniques along with the SRT algorithm. These requirements increase with the increase of the radix-n number; thus, SRT algorithm implementation is restricted to fewer than ten numbers. A very high radix generally refers to an SRT algorithm that retires more than 10 bits in one iteration. The basic difference between the SRT and high radix algorithm is the different logic of quotient bit selection and multipliers' number and width. An increase in radix causes the use of a quotient bit selection logic table that is impracticable in size, which ultimately affects the cycle time. Approximation and pre-scaling techniques do not require an extra multiplier.

Unlike the SRT algorithm, Svoboda gave an alternative possibility to generate quotient bit selection logic based on only a partial remainder. Thus the criticality of the quotient bit selection logic gets reduced as compared to the SRT algorithm. Although the generalized Svoboda algorithm gives shorter quotient bit selection logic, it required normalized and pre-scaled operands; otherwise, it utilizes extra two multipliers causing more area and time. Later in the new Svoboda-Tung algorithm developed with a signed digit number system which avoids overshoot due to compensation, by implementing the alternative method of recoding two MSB's of the partial remainder with alternate consecutive positions causing to follow and keep the partial remainder in bounded condition. Svoboda-Tung algorithm is valid for radix more than two, whereas the new Svoboda-Tung algorithm is valid for generalized radix range. In the case of the functional

TABLE 13. (a) Summary of different division algorithms.

Sr. No.	Algorithm	Equations	Important Points
1	Long Division	For J^{th} iteration $q_j = 0$ if $2R_{j-1} < D_r$ $q_j = 1$ if $2R_{j-1} \geq D_r$ $R_j = 2R_{j-1} - q_j \times D_r$	It is similar to a normal paper and pencil algorithm
			Digit recurrence class
			Iterative subtraction is performed
			Only a single quotient bit is calculated in each iteration
			No requirement for a look-up table
			The shift register, subtractor, multiplier for every iteration gives the approximate area requirement for algorithm implementation
2	Restoring	For J^{th} iteration $q_j = 0$ if $R'_j < 0$ $q_j = 1$ if $R'_j \geq 0$ $R_j = 2R_{j-1}$ if $q_j = 0$ $R_j = R'_j$ if $q_j = 1$ $R'_j = 2R_{j-1} - D_r$	It is similar to the long division algorithm
			Simple logic for implementation
			No requirement for a look-up table
			Iterative subtraction is performed
			The non-redundant number system is used to write a quotient.
			If the partial remainder value comes other than positive or zero, then the divisor is restored by the subtraction result performed in that iteration
			It requires a full-width comparator in each iteration, and subtractor, shift register, multiplier gives the approximate area requirement for algorithm implementation
3	Non restoring	For J^{th} iteration $q_j = -1$ if $R_{j-1} < 0$ $q_j = 1$ if $R_{j-1} \geq 0$ $R_j = 2R_{j-1} + D_r$ if $q_j = -1$ $R_j = 2R_{j-1} - D_r$ if $q_j = +1$	Similar to restoring algorithm and it does not require to restore the partial remainder if subtraction goes negative
			No requirement for a look-up table
			Operation in each iteration depends on the result of the previous iteration.
			Only one addition or subtraction can be performed in each iteration, so separate hardware is required
			Partial remainder kept between $-D_r$ to $+D_r$ and quotient digit -1 or 1
			It requires a sign bit to decide whether to perform either addition or subtraction; adder, subtractor, and shift register gives the approximate area requirement for algorithm implementation
4	SRT	For J^{th} iteration $q_j = \bar{1}$ if $2R_{j-1} < -D_r$ $q_j = 0$ if $-D_r \leq 2R_{j-1} \leq D_r$ $q_j = 1$ if $2R_{j-1} \geq D_r$ $\frac{1}{2}(n-1) \leq m \leq n-1$ $n = 2^b$ and $k = x/b$ $Q = \sum_{j=1}^k q_j n^{-j}$	It is also non restoring algorithm based on radix-n
			Named after Dura W. Sweeney, James E. Robertson, and Keith D. Tocher
			For x bit, integer division requires $k=x/b$ iterations, $b=$ number of bits detected in each iteration
			n decides how many quotient bits are to be detected in each iteration; if $n=2$, then one quotient bit is detected per iteration, radix-n is typically selected as a power of base 2
			Each quotient digit has a value from $\{-m, -m+1, \dots, -1, 0, 1, \dots, m-1, m\}$
			The algorithm implements 2's complement value of D_r instead of $-D_r$, which indeed provides shifting over zeros to eliminate extra adder and subtractor
			Needs extra subtractor to find out next partial remainder
			Error results due to few MSB's being used to predict quotient bits as in low radix, which decreases with the increase of radix
			Requires quotient selection look-up table. Quotient select table plus carry-save adder (CSA) gives the approximate area requirement for algorithm implementation and shows the iteration time of accessing quotient select table plus multiple form and subtraction
5	Very high radix	*****	It uses multiplication to form divisor multiples
			A look-up table is required for obtaining an initial approximation to reciprocal and quotient digit selection logic
			Differs from normal radix-n divider in terms of number and type of operations used in each iteration and quotient digit selection logic

TABLE 13. (Continued.)(b) Summary of different division algorithms.

6	Svoboda Algorithm And Svoboda-Tung Algorithm	$\left\{ \frac{mn}{(m+1)(n-1)} < D_r < \frac{m(n-2)}{(n-1)(m-1)} \right\}$ $\{-m/n-1 < R_j < m/n-1\}$ $Range = \{0, \pm 1, \dots, \pm m\}$ $Boundary\ limit = \{n/2 + 1 \leq m \leq n - 1\}$	<p>Quotient digit is predicted based on the partial remainder without considering divisor; one or two MSBs of the partial remainder are used for generating quotient digit selection logic</p> <p>It can select quotient digit out of the radix range as an overflow occurred due to compensation</p> <p>It requires pre-scaled operands and can work on conventional and signed digit number range</p> <p>Like the SRT algorithm, it is also a radix-n based algorithm</p>
7	New Svoboda-Tung Algorithm	$SBD = \{-1 \leq m \leq 1\}$ $D_r\ range = \{0, 1, \dots, n - 1\}$ $Boundary\ limit = \{-m/n - 1 < R_{j+1} < m/n - 1\}$	<p>It records/re-stores the partial remainder's two most significant digits to overcome overflow due to compensation</p> <p>This arrangement allows for addition/subtraction with carry propagation up to one left position</p> <p>Rest the working is similar to that of the Svoboda-Tung algorithm</p> <p>Total conversion time is given as a collective time required for scaling, recursion, conversion in terms of an initial clock cycle, thus $T_{div} = (T_{scale} + T_{rec} + T_{conv}) * T_{clk}$</p> <p>Quotient digit selection logic, one full word length fast carry propagation adder, 2 full word length carry free adder/subtractor, and 2 full word length latches gives the approximate area requirement for algorithm implementation depending on the selection of maximal or minimal redundancy</p> <p>The major drawback of this algorithm is high hardware overhead</p>
8	Look-up table	*****	<p>It stores the values, logic, numbers, quotients, etc., which are useful to execute division</p> <p>Look-up table class algorithm can be utilized along with functional iterative class and high radix algorithms</p> <p>Look-up tables are useful in initial approximation in the case of SRT and functional iteration class algorithms.</p> <p>A direct approximation, linear approximation, partial product array, pre-computed values array, result, or quotient cache are the common examples of look-up table implementation in the division algorithm</p>
9	Newton-Raphson	$Q = D_d/D_r = p \times (q)^{-1}$ $f(X) = 1/X - q^{-1} = 0$ $X_{i+1} = X_i - \frac{f(X_i)}{f'(X_i)}$ $X_{i+1} = X_i - \frac{(1/X_i - q^{-1})}{-1/X_i^2} = X_i \times (2 - q^{-1} \times X_i)$ $\epsilon_{i+1} = \epsilon_i^2 (q^{-1})$	<p>It comes under the functional iteration class</p> <p>Requires look-up table</p> <p>It works on the estimation technique</p> <p>It considers the convergence of quotient by estimation or prediction</p> <p>The final quotient is derived by multiplying approximated reciprocal and dividend</p> <p>Shows error due to inaccuracy of quotient digit prediction or estimation</p> <p>It requires multiplication and addition or subtraction at each iteration</p> <p>The accuracy can be improved by selecting a proper root at the beginning</p> <p>Latency and error in convergence are directly dependent on the root selected at the beginning of the convergence and shows the iteration time approximately equals to the time required for two serial multiplication</p> <p>Multiplier, quotient select look-up table, and control logic gives the approximate area requirement for algorithm implementation</p>

TABLE 13. (Continued.) (c) Summary of different division algorithms.

10	Series Expansion	$g(y) = g(p) + (y - p)g'(p) + \frac{(y - p)^2}{2!} g''(p) + \dots$ $q = a/b = a \times g(y)$	It comes under the functional iteration class and shows the possibility to use pipeline or parallel architecture of hardware
			Series represents the root of the anti-divisor or reciprocal, which can be used in the iterations for approximation
			Each iteration performs prescaling dividend and divisor by series approximation or rounding off and then performs series convergence
			Multiplier, quotient select look-up table, and control logic gives the approximate area requirement for algorithm implementation and shows the iteration time approximately equals the time required for multiplication or two multiplication in parallel
11	Variable Latency	*****	Variable execution time thus results in different conversion time for a different set of dividend and divisor
			The DEC Alpha 21164 is one of the best examples of variable latency class algorithm implementation, which is based on the concepts of the simple normalizing non-restoring division algorithm
			Self-timing, result cache, and speculation of quotient digit are some of the techniques used for providing variable latency
12	Goldschmidt	$D_d/D_r = N/D = A/B$ $x_{n+1} = x_n(2 - y_n) = x_n r_n$ $y_{n+1} = y_n(2 - y_n) = y_n r_n$	It is a convergence based functional iterative class divider algorithm
			It multiplies both dividend and divisor by anti-divisor or reciprocal
			It originates from the Taylor-Maclaurin series of $1/(x + 1)$
			It does not provide a remainder
			1's complement can be used instead of $(2 - y_n)$ to avoid carry propagation delay, but it adds a new approximation error in each iteration
			Quotient digit selection logic look-up table, one full word length multiplier, and one full word length adder/subtractor logic gives the approximate area requirement for algorithm implementation
13	Taylor Series	$q = D_d/D_r \text{ and } X_0 = 1/D_r$ $q = D_d X_0 \left\{ \begin{matrix} 1 + (1 - D_r X_0) + (1 - D_r X_0)^2 \\ + (1 - D_r X_0)^3 \end{matrix} \right\}$	It is also a multiplicative iteration based algorithm
			The precision depends upon the closeness with anti-divisor (reciprocal) estimation
			It provides a parallel powering section that computes high order terms faster with minimal extension to hardware overhead
			Quotient digit selection logic look-up table, and three full word length multiplier gives the approximate area requirement for algorithm implementation
14	Smaller Dividend	$N_1 = \sum_{i=0}^{2n-1} x_{2n+i} 2^{2n+i}$ $N_2 = \sum_{i=0}^{2n-1} x_i 2^i$ $D_d = N_1 + N_2$ $D_d/D_r = (N_1 + N_2)/D_r = N_1/D_r + N_2/D_r$	It is the simplest parallel computing algorithm
			The basic phenomenon behind this algorithm is to consider division as a fraction
			It requires an actual dividend greater than the divisor, i.e., dividend bit count as $4n$ and divisor bit count as n
			We can represent dividends in terms of fixed partitions based on associated weights as per the dividers' radix
			The area is directly dependent on the number of dividend partitions related to the dividers' radix
15	Jebelean Exact Division	$D_d = d * Q$ $D_d = D_{dupk} n^k + D_k$ $b_k = (-n^{-k} D_{dk}) \text{ mod } d$ $b_k = (-n^{-k} \text{ mod } d D_{dk}) \text{ mod } d$	It is applicable when completed division is performed on long integer operands in digital computation even after knowing that the remainder will be zero
			It works from the least significant digit of the operands
			Remarkable performance is observed when radix is prime or power of 2
			It takes constant execution time to access a fixed word length look-up table
			It takes $O(\log n)$ execution time, and for short division, $O(n/\rho + \log \rho)$, where n is the word length of dividend and ρ is the number of processors

TABLE 14. (a) Comparison table.

Long Division	Simple and similar to paper and pencil algorithm	Only one quotient bit can be detected in each iteration	For x bit dividend x shifts are required and subtraction	x where x is the number of bits in input operands	Area utilization of implementation is dependent on the size of operands as the number of subtractive iteration depends on divisor and dividend size. One iteration area utilization is approximately equal to the area required to implement shift register, subtractor, and comparator
	Simplest conversion logic	Special test condition is required to check if dividend is greater than divisor. Possibility of loss of most significant bit causing error			
	No look-up table required	Area and latency inefficient			
	Remainder is not required after last iteration so avoids final subtraction	The comparison to determined qi before subtraction to determine new partial remainder			
Restoring	Simple implementation similar to long division algorithm	Possible loss of most significant bit (MSB)	For x bit dividend x shifts are required and subtraction	x where x is the number of bits in input operands	Area utilization of implementation is approximately equal to the area required to implement subtractor, shift register, multiplier, and comparator
	Less complex conversion method	Check for overflow is required			
	No extra test is required when dividend is less than divisor	Execution is slower as it requires restoration of remainder in each iteration			
	The remainder and quotient values remain either positive or zero	Requires separate registers for partial remainder in each iteration			
	The divisor is added back to result of division when divisor subtraction produces a negative result in iteration	Requires full width comparison at every iteration to get one bit of quotient			
		To perform division it requires to have positive dividend and divisors Quotient needed to be rearranged to get actual quotient			
Non restoring	Doesn't restore partial remainder	Requires extra bit to be added with partial remainder to have a track on sign	For x bit dividend x shifts are required and subtraction / addition	x where x is the number of bits in input operands	Area utilization of implementation is approximately equal to the area required to implement adder, subtractor, and shift register
	Need to check only sign bit of partial remainder				
	Quotient is the actual quotient that we required	Requires separate adder and subtractor in each iteration			
	Can be improved by replacing subtraction by adding 2's complement				
Very high radix	Reduces iteration and thus latency	High radix makes quotient selection logic more complex and impractical to implement Demand to use very large look up tables	Approximatly time required to access quotient select table, multiple form, and subtraction	$\left\{ \frac{x}{n} \right\} + Scale$ where x is the number of bits in input operands	Area utilization of implementation is approximately equal to the area required to implement quotient select table and carry-save adder (CSA)
Goldschmidt	It is a convergence based functional iterative class divider algorithm; It provides quadratic convergence for anti-divisor and division operation	The algorithm's main drawback is that it does not yield a remainder, limiting its application only for the floating-point implementation	Approximatly time required to access quotient select table three multipliers, and two adders	$\left\{ \log_2 \frac{x}{j} + 1 \right\} t_{mul} + 2$ if $t_{mul} > 1$	Area utilization of implementation is approximately equal to the area required to implement one full word length multiplier, adders, and quotient select table; Approximate area depends on the architecture considered to use a multiplier, i.e., three in series or one in sharing, i.e., reuse
	It multiplies both dividend and divisor by anti-divisor or reciprocal	As it requires multipliers, which are fast but larger in area		$\left\{ 2 \left[\log_2 \frac{x}{j} \right] + 3 \right\}$ if $t_{mul} = 1$	
	It originates from the Taylor-Maclaurin series of $1/(x + 1)$	1's complement can be used instead of (2 - y_n) to avoid carry propagation delay, but it adds a new approximation error in each iteration		and approximately 16-18 cycles	
	It can imply fast multiply schemes than carry propagate adders	The floating-point multiplier can be shared between iterative multiply operations to reduce area but results in long latency, and subsequent operation can't be started until the previous operation ends			
	Due to two parallel multipliers, it can provide more parallelism than other functional iterative algorithms				
Taylor Series	The precision depends upon the closeness with anti-divisor (reciprocal) estimation	As it requires multipliers that are fast but larger in area	Approximatly time required to access one look-up table, three multiplication and three adder/subtraction	8-12 cycles	Area utilization of implementation is approximately equal to the area required to implement quotient select table and three full word length multipliers
	It provides a parallel powering section like squaring and cubing section that computes high order terms faster with minimal extension to hardware overhead	Precision depends on the closeness with initial approximation as it requires to take several iterations to reach the required precision			
Svoboda and Svoboda-Tung	Quotient digit is predicted based on the partial remainder without considering divisor. One or two MSBs of the partial remainder are used for generating quotient digit selection logic	It can select the quotient digit out of the radix quotient digit range as an overflow occurred due to compensation	$T_{div} = (T_{scale} + T_{rec} + T_{conv}) * T_{clk}$	*****	Area utilization of implementation is approximately equal to the area required to implement quotient select table, 1 full word length fast carry propagation adder, 2 full word length carry free adder/subtractor +2 full word length latches
	It requires pre-scaled operands and can work on conventional and signed digit number range	If input operands are not pre-scaled, then it requires two extra multipliers It is applicable more than radix 4			
New Svoboda-Tung	It is similar to that of the Svoboda and Svoboda-Tung algorithm, except it overcomes the overflow problem due to compensation. It uses recording two consecutive MSB's of the partial remainder	Pre-scaled operands are required else; it needs extra multipliers resulting in more hardware overhead	$T_{div} = \left(3 + \frac{W}{2} \right) + T_{conv} * \left(22 + \frac{W}{4} \right)$	*****	Area requirement is appriximally area required by LUT, 1 full word length fast carry propagation adder, 2 full word length carry free adder/subtractor, and 2 full word length latches
Jebelean Exact Division	It works from the least significant digit of the operands, making it more suitable for parallel architecture implementation like GPU or MIC	It is applicable for exact division, knowing that the remainder will be zero	$O(\log n)$ and $O(n/\rho + \log \rho)$ for short division	*****	Depends on GPU or MIC architecture
	Remarkable performance is observed when radix is prime or power of 2. It takes constant execution time to access the fixed word length look-up table	Borrow calculation in parallel is challenging and critical, which needs to follow synchronization requirements			

TABLE 14. (Continued.) (b) and (c) Comparison table.

SRT	It produces a fixed bit of quotient in each iteration	The choice of selecting higher quotient bits causes complexity in quotient selection logic	Approximately time required to access quotient select table, multiple form, and subtraction	$\left\{ \left\lceil \frac{x}{n} \right\rceil + Scale \right\}$ where x is the number of bits in input operands and n is the radix	Area utilization of implementation is approximately equaled to the area required to implement quotient select table and carry-save adder (CSA)
	It is an improvement over a non-restoring algorithm	Higher radix implementation is difficult due to impractical multiples of the divisor			
	Doesn't require a separate adder and subtractor, unlike a non-restoring algorithm	To overcome these problem requires the use of pre-scaling and prediction method which increases overhead			
	Determines more than one quotient bit	Needs to convert the last remainder to conventional representation to find out sign bit			
	Reduce latency time by increasing radix	Rounding provisions are required, generally performed by computing an extra digit, i.e., guard digit in quotient and examining it in the final remainder			
		Need to normalize divisor prior to starting division requires extra hardware			
	It requires extra multipliers for high radix-n stages, causing increased access time and increases the criticality and size of a look-up table				
	Quotient correction stage selection is dependent on the sign bit				
Newton-Raphson	It can select any one root of priming function from available roots	Use of 1's complement includes more error	Approximately time required to access two serial multiplication and subtraction	$\left\{ 2 \left\lceil \log_2 \frac{x}{j} \right\rceil + 1 \right\} t_{mul} + 1$ where x is the number of bits in input operands and j is the number of bits of accuracy from initial approximation and t _{mul} is the latency of multiplier fused adder unit	Area utilization of implementation is approximately equaled to the area required to implement quotient select table, one multiplier, and control logic
	Each iteration contains two multiplications and subtraction				
	Subtraction can be performed as 2's complement				
	It can also use 1's complement to reduce area and timing				
Series expansion	Series represents the root of the anti-divisor or reciprocal, which can be used in the iterations for approximation	Iteration operations are independent; error in one iteration is not self-corrected in the next iteration	Approximately time required to perform one-two multiplication	$\left\{ \left\lceil \log_2 \frac{x}{j} \right\rceil + 1 \right\} t_{mul} + 2 \text{ if } t_{mul} > 1$ $\left\{ 2 \left\lceil \log_2 \frac{x}{j} \right\rceil + 3 \right\} \text{ if } t_{mul} = 1$ where x is the number of bits in input operands and j is the number of bits of accuracy from initial approximation and t _{mul} is the latency of multiplier fused adder unit	Area utilization of implementation is approximately equaled to the area required to implement quotient select table, one to two multiplier, and control logic
	Each iteration performs pre-scaling dividend and divisor by series approximation or rounding off and then performs series convergence	Shows rounding error in multiplications sums through the iterations			
	First, prescale dividend and divisor by initial approximation followed by direct convergence to the quotient	Requires wide bit size multiplier			

iteration algorithm, the quotient convergence is quadratic, which works on the initial approximation. In the Newton-Raphson algorithm, which shows the result in the product term of dividend and reciprocal, the reciprocal depends on the selection of the priming function, which points out its root at the reciprocal or anti-divisor, which generally has many values.

Based on which root is selected, quotient convergence accuracy will vary, causing an error in the division and generating overhead if the root selected is over the true quotient. It means that the multiplication is dependent and must be performed sequentially. In series expansion, iteration performs pre-scaling of the dividend and divisor by series approximation or rounding off and then performs series convergence; thus, the multiplication can be implemented in parallel. The functional iteration algorithm does not provide the final remainder at the last iteration. In the variable latency class, self-timing, result cache, and quotient digit speculation techniques have been used to provide reduced average latency. A reciprocal cache can be utilized effectively along with a functional iterative algorithm to reduce the time required for initial approximation. An additional area will be required for implementing a reciprocal cache, but it will be less than that

required for the initial approximation look-up table. The self-timing technique requires the use of switching techniques, which can clock the circuit synchronously with the other components of the algorithm along with test checking to confirm correct operation. The division algorithm can be implemented in three hardware architectures: serial, pipelined, and parallel. Serial and pipelined architecture implementation of the division algorithm is comparatively slower than parallel implementation but more area efficient than parallel implementation. Synchronization of various divider units is the main problem associated with parallel architecture, which can be critical due to the sluggish behavior of hardware components used in a parallel architecture over time. The generalized application like CPU, FPGA, ASIC serial, and pipeline dividers are prone to be used due to their less area and controlling requirements, whereas critical applications like Graphics Processing Unit (GPU) and Many Integrated Cores (MIC) require the fastest implementation of dividers, so parallel dividers are preferred over serial and pipelined dividers. Table 13 (A), Table 13 (B), and Table 13 (C) give a summary of the different division algorithms. Table 14 (A) and Table 14 (B) illustrates a summary of the comparative study considering the approximate iteration time, latency, and area.

XI. CONCLUSION

The division is the most complex basic arithmetic operation, and efforts have been made to improve its implementation in digital circuits, computer systems, and embedded systems by optimizing the area, hardware resources needed, or latency cycles. Generally, improvement in one of those aspects worsens the others; thus, one must select a particular technique based on the specific application requirements, which gives room for continuing research on developing an algorithm for division operations suitable for new generation application requirements. For the implementation of division operations in the area concerning portable programmable devices, FPGAs are vital because of the emerging applications in which these devices are used to implement some critical system-on-chip application or improve the existing application, and the results of indirect division operation are not sufficient. As explained in the article, restoring and some non-restoring algorithms implement simple conversion logic but require a long time and large area. Although the conversion logic is simple, it does not suit high-frequency applications due to latency problems. Also, in the case of sensor nodes, portable devices of the IoT, where the area is of major concern, it fails due to the large area requirements for implementing these algorithms. However, restoring and non-restoring algorithms are the main point of study for developing new algorithms to perform division operations theoretically and electronically. The radix-based SRT division algorithm is one of the most implemented non-restoring algorithms. Although the SRT algorithm was the first choice for commercial implementation in the majority of soft and modern processors like Intel's Pentium processor, FPGAs controllers, and ALU units of complex hardware, it is restricted to certain low radix values, especially less than 10. Radix-2 and radix-4 are the most implementable formats of the SRT algorithm. The main reasons for restricting SRT algorithm implementation to certain low radix values are the increase in the quotient selection logic's criticality and the enormous increase in area requirements for storing look-up tables for this logic. This causes it to fail to follow the execution cycle, which is considered as two cycles. Whereas low radix implementation provides low area requirements and possibly follows very tight conditions of execution cycle time, its major drawback is the higher latency, which depends on the number of bits discovered in every iteration; in low radix implementation, this is restricted to one or two quotient bits per iteration. Therefore, to reduce division latency, more bits need to be retired in every cycle. However, directly increasing the radix can improve the cycle time at the cost of increasing the complexity of divisor multiplier formations. The alternative is a pipelined structure or two-stage lower radix stages combined to form higher radix dividers by simple staging or possibly overlapping one or both the quotient selection logic and partial remainder computation hardware.

Svoboda algorithm is also another radix based divider algorithm. The quotient bit is generated in the Svoboda algorithm

based on the only partial remainder, unlike the SRT algorithm. The quotient bit selection logic is based on partial remainder and divisor. Although the Svoboda algorithm uses the only partial remainder for quotient bit selection logic, it requires normalized and pre-scaled operands. If not, then it requires an extra two multipliers causing more area and time requirement. The pre-scaled divisor needs to be in a certain range near to 1. Thus, it can be represented as $(1+e_r)$, where e_r is a small positive fractional value $e_r < 1/n$ and n is the radix. In each iteration, if q_j results in $-ve$, it indicates overshooting, to compensate overshooting by adding/subtracting e_r and performing right shift operation by $j-1$ places depending on the last step was subtraction/addition. A new Svoboda-Tung algorithm is presented with a sign digit range to overcome the Svoboda algorithm's limitations. The major drawback of the new Svoboda-Tung algorithm is that it generates a direct quotient value, but the final remainder should be calculated by scaling partial remainder with the same factor as the operands. Thus it restricts its use with applications where the unscaled remainder is a must.

All these radix base alternatives lead to an increase in the area, conversion complexity, and potentially the cycle time. In contrast, the functional iterative class offers an alternative to the SRT algorithm. It computes the quotient bit based on estimation or approximation of series expansion functions like Neuton-Rapson Goldschmidt, Taylor series, etc. It utilizes multiplication instead of subtraction operations, which ultimately reduces the number of iterations and can generate multiple quotient digits in one iteration with low latency. The use of multiplication for functional iteration dividers makes it more complex than simple digit recurrence dividers. This type of divider has a major drawback of the quotient bit's inaccuracy because of direct rounding off of the approximate solution values rather than infinitely precise values. The error depends on the accuracy of the initial estimation. In the Newton-Raphson iteration, which is limited to two multiplications and must proceed in series, a large error is generated. Reducing the error requires the introduction of a trade-off between the additional chip area for the look-up table and the latency of the divider. The series expansion provides relatively lower latency. The area-focused implementation refers to shared multipliers and creates an additional enmity for the multiplier, which can be overcome by an additional multiplier, causing an area increase. Goldschmidt algorithm is another functional iterative divider that multiplies both dividend and divisor by anti-divisor, whereas in Neuton-Rapson, it multiplies only with the dividend. This algorithm's major drawback is that it does not provide the remainder, making it useful only for the floating-point division [109]. First multiplication required for finding out values of x_n , and y_n requires full precision. Another drawback, 1's complement can be used instead of $(2 - y_n)$ to avoid carry propagation delay, but it adds a new approximation error in each iteration. In Taylor series dividers, Taylor series expansion calculates accurate anti-divisor (reciprocal) to reduce the error in the

least important bits of quotient precision with a parallel powering section that computes high-order terms caused extra hardware overhead causing area increase.

Variable latency class dividers are very rare due to their complexity and area constraints. Some techniques like radix- n , functional iteration, and variable latency require extra storage for look-up tables, and the high radix reduces the latency but requires a large capacity look-up table, which is impractical for implementation. The look-up table requires storage like ROM, which increases the area requirements for implementation. Optimized area and hardware resources are needed, or the latency cycles need to be interrelated. There are three possibilities of the hardware architecture that can be used for dividers implementation. Serial hardware architecture, generally maximum division algorithms, processes sequentially, so it is best suited for implementation. However, the sequential implementation provides less area and easy logic for implementation but requires higher latency and conversion time, making it infelicitous for highly critical applications. The parallel hardware architecture is contrasting with serial architecture. Parallel architecture has several same element configuration devices or cores connected in parallel, simultaneously operating, causing a reduction in latency and execution time. As it requires multiple cores to work together simultaneously, it makes critical synchronization and high area requirements, leading to increased implementation cost. Thus parallel architecture implementation is costlier, making it unique for critical applications like graphics processing units (GPU). A pipelined architecture is the best choice for achieving parallelism in sequential architecture with parallel processing. Some or all processes of division algorithms can be pipelined to achieve partial parallel processing. GPU and MIC have an advantage of parallel architecture for achieving low latency and execution time on account of the high area and complex controlling logic. As the division algorithm's initial nature is sequential, GPU and MIC require developing a complex controlling logic to ensure parallelism requirements. In Jebelean exact division algorithm, we have an experience that the simultaneous borrow calculation is quite critical, and in Takahashi's algorithm, the remainder is executed sequentially, and if the remainder could get parallelly, then the final quotient could get in parallel. Thus Takahashi uses a parallel cyclic reduction method to solve the remainder recurrence. Division algorithm implementation on parallel architecture can cost large hardware overhead due to the use of multiple cores in parallel, which ultimately leads to high implementation cost but quicker execution.

On the contrary, the CPU incorporates sequential or pipelined architecture to imply division algorithms with less complexity and hardware overhead on account of latency and execution time, which can be improved to some extent by using variable latency division algorithms. Thus the use of CPU based implementation is very useful and suitable for general purpose and dedicated embedded applications, an ASIC or FPGA based applications where the area is more concerned. On the other hand, applications with a high-speed

response as the first priority and area as a second priority can use GPU and MIC implementation like graphics processing, biomedical applications, artificial intelligence, research applications, etc. The use of architecture is not limited or restricted to a particular application. Maximum division algorithms can be implemented by serial, parallel, or pipelined architecture depending on cost, area, and complexity suitability with the application. Generally, improvement in one of those aspects worsens the others; thus, one has to select a particular algorithm based on the specific application requirements. This opens the possibility of developing a new technique or combination of techniques, which are fast in operation and area-efficient.

XII. FUTUR WORK

Based on the review, it is found out that the digit recurrence division algorithm is most likely preferred for implementation in different applications considering its ease in conversion logic and considerable area and latency constraints. Area and latency constraints are very important for embedded systems and ASIC design, where fast action in a considerably less area is of great importance. We put efforts into developing a new digit recurrence algorithm, which has simple conversion logic and benefits in the area and variable conversion time constraints of the divider circuit. The target for future works

1. To improve the basic idea of a new algorithm by standardising algorithm steps to achieve area and timing improvements.
2. Floorplanning and circuit implementation using multiple logic families for delay and power consumption comparison.
3. Utilizing a new circuit in different applications to verify results.

ACKNOWLEDGMENT

A preliminary patent is applied in Estonia based on the research work of developing a new algorithm for division. Application no-70390 date-June 2020.

REFERENCES

- [1] *Merriam-Webster Dictionary*. Accessed: Jul. 2020. [Online]. Available: <https://www.merriam-webster.com/dictionary/mathematics>
- [2] *Cambridge Dictionary by Cambridge University Press*. Accessed: Jul. 2020. [Online]. Available: <https://dictionary.cambridge.org/dictionary/english/mathematics>
- [3] R. K. L. Trummer, "A high-performance data-dependent hardware integer divider," M.S. thesis, Inst. Comput. Sci. Syst. Anal., Paris Lodron Univ., Salzburg, Austria, May 2005.
- [4] D. G. Bailey, "Space efficient division on FPGAs," in *Proc. Electron. New Zealand Conf.*, 2006, pp. 206–211.
- [5] J. Kumari and M. Y. Yasin, "Design and Soft Implementation of N-bit SRT Divider on FPGA through VHDL," *Int. J. Innov. Eng., Sci. Manage.*, vol. 3, no. 4, pp. 13–19, Apr. 2015.
- [6] K. Narendra, S. Ahmed, S. Kumar, and G. H. Asha, "FPGA implementation of fixed point integer divider using iterative array structure," *Int. J. Eng., Tech. Res.*, vol. 3, no. 4, pp. 170–179, Apr. 2015.
- [7] E. Matthews, A. Lu, Z. Fang, and L. Shannon, "Rethinking integer divider design for FPGA-based soft-processors," in *Proc. IEEE 27th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2019, pp. 289–297, doi: 10.1109/FCCM.2019.00046.

- [8] K. Tatas, D. J. Soudris, D. Siomos, M. Dasygenis, and A. Thanailakis, "A novel division algorithm for parallel and sequential processing," in *Proc. 9th Int. Conf. Electron., Circuits, Syst.*, Dubrovnik, Croatia, Sep. 2002, pp. 553-556.
- [9] K. D. Tocher, "Techniques of multiplication and division for automatic binary computers," *Quart. J. Mech. Appl. Math.*, vol. 11, no. 3, pp. 364-384, 1958.
- [10] H. Asai, "A recursive radix conversion formula and its application to multiplication and division," *Comput. Math. with Appl.*, vol. 2, nos. 3-4, pp. 255-265, 1976.
- [11] N. Sorokin, "Implementation of high-speed fixed-point dividers on FPGA," *J. Comput. Sci. Technol.*, vol. 6, no. 1, pp. 8-11, Apr. 2006.
- [12] A. Kaplan, *Math on Call: A Mathematics Handbook*. Wilmington, MA, USA: Great Source Education Group, 2004.
- [13] T. Bassarear and M. Moss, *Mathematics for Elementary School Teachers*, 4th ed. Independence, KY, USA: Cengage Learning, 2008.
- [14] S. F. Obermann and M. J. Flynn, "Division algorithms and implementations," *IEEE Trans. Comput.*, vol. 46, no. 8, pp. 833-854, Aug. 1997.
- [15] S. Dixit and M. Nadeem, "FPGA accomplishment of a 16-bit divider," *Imperial J. Interdiscipl. Res.*, vol. 3, no. 2, pp. 140-143, 2017.
- [16] M. F. Kasim, T. Adiono, M. F. Zakiy, and M. Fahreza, "FPGA implementation of fixed-point divider using pre-computed values," in *Proc. Technol.*, vol. 11, Jun. 2013, pp. 206-211, doi: 10.1016/j.protcy.2013.12.182.
- [17] G. Sutter, G. Biol, and J.-P. Deschamps, "Comparative study of SRT-dividers in FPGA," in *Field Programmable Logic and Application (Lecture Notes in Computer Science)*, vol. 3203, J. Becker, M. Platzner, and S. Vernalde, Eds. Berlin, Germany: Springer, 2004, pp. 209-220.
- [18] R. S. Hongal and D. J. Anita, "Comparative study of different division algorithms for fixed and floating point arithmetic unit for embedded applications," *Int. J. Comput. Sci. Eng.*, vol. 4, no. 9, pp. 48-54, 2016.
- [19] S. Kaur, M. Singh, and R. Agarwal, "VHDL implementation of non-restoring division algorithm using high-speed adder/subtractor," *Int. J. Adv. Res. Electr., Electron. Instrum. Eng.*, vol. 2, no. 7, pp. 3317-3324, Jul. 2013.
- [20] N. Boullis and A. Tisserand, "On digit-recurrence division algorithms for self-timed circuits," INRIA-Institut Nat. De Recherche En Informatique Et En Automatique, France, Tech. Rep. RR-4221, Jul. 2001.
- [21] J. E. Robertson, "A new class of digital division methods," *IRE Trans. Electron. Comput.*, vol. 7, no. 3, pp. 218-222, Sep. 1958.
- [22] D. Wong and M. Flynn, "Fast division using accurate quotient approximations to reduce the number of iterations," *IEEE Trans. Comput.*, vol. 41, no. 8, pp. 981-995, Aug. 1992.
- [23] S. F. Oberman and M. J. Flynn, "Design issues in division and other floating-point operations," *IEEE Trans. Comput.*, vol. 46, no. 2, pp. 154-161, Feb. 1997.
- [24] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers, "The IBM System/360 model 91: Floating-point execution unit," *IBM J. Res. Develop.*, vol. 11, no. 1, pp. 34-53, Jan. 1967.
- [25] D. L. Fowler and J. E. Smith, "An accurate, high speed implementation of division by reciprocal approximation," in *Proc. 9th IEEE Symp. Comput. Arithmetic*, Sep. 1989, pp. 60-67.
- [26] X. Fang and M. Leeser, "Open-source variable-precision floating-point library for major commercial FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, no. 3, p. 20, Jul. 2016, doi: 10.1145/2851507.
- [27] Xilinx Inc. *MicroBlaze Processor Reference Guide*. Accessed: Aug. 2020. [Online]. Available: <https://xilinx.com/support/documentation/swmanuals/xilinx20164/ug984-vivado-microblaze-ref.pdf>
- [28] Intel Corp. *Nios II Gen2 Processor Reference Guide*. Accessed: Aug. 2020. [Online]. Available: https://altera.com/en_US/pdfs/literature/hb/nios2/n2cpu-nii5v1gen2.pdf
- [29] *GRLIB IP Core User's Manual, Cobham Gaisler AB*. Accessed: Aug. 2020. [Online]. Available: <https://gaisler.com/products/grlib/grip.pdf>
- [30] J. Liu, M. Chang, and C.-K. Cheng, "An iterative division algorithm for FPGAs," in *Proc. Int. Symp. Field Program. Gate Arrays (FPGA)*, Monterey, CA, USA, 2006, pp. 83-89.
- [31] A. A. Varghese, C. Pradeep, M. E. Eapen, and R. Radhakrishnan, "FPGA implementation of area-efficient IEEE 754 complex divider," in *Proc. Technol.*, vol. 24, 2016, pp. 1120-1126, doi: 10.1016/j.protcy.2016.05.245.
- [32] D. L. Harris, S. F. Oberman, and M. A. Horowitz, "SRT division architectures and implementations," in *Proc. 13th IEEE Symp. Comput. Arithmetic*, Asilomar, CA, USA, Jul. 1997, pp. 18-25, doi: 10.1109/ARITH.1997.614875.
- [33] Sumiksha, P. Konda, and S. Shetty, "Computation of SRT and CORDIC division algorithms," *IOSR J. Electron. Commun. Eng.*, vol. 12, no. 4, pp. 53-56, July/Aug. 2017.
- [34] S. Oberman, "Design issues in high-performance floating-point arithmetic units," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Stanford Univ., Stanford, CA, USA, Nov. 1996.
- [35] M. D. Ercegovic and T. Lang, "Simple radix-4 division with operands scaling," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1204-1208, Sep. 1990.
- [36] J. Fandrianto, "Algorithm for high speed shared radix 8 division and radix 8 square root," in *Proc. 9th Symp. Comput. Arithmetic*, Jul. 1989, pp. 68-75.
- [37] S. E. McQuillan, J. V. McCanny, and R. Hamill, "New algorithms and VLSI architectures for SRT division and square root," in *Proc. IEEE 11th Symp. Comput. Arithmetic*, Jul. 1993, pp. 80-86.
- [38] P. Montuschi and L. Ciminiera, "Reducing iteration time when result digit is zero for radix 2 SRT division and square root with redundant remainders," *IEEE Trans. Comput.*, vol. 42, no. 2, pp. 239-246, Feb. 1993.
- [39] P. Montuschi and L. Ciminiera, "Over-redundant digit sets and the design of digit-by-digit division units," *IEEE Trans. Comput.*, vol. 43, no. 3, pp. 269-277, Mar. 1994.
- [40] P. Montuschi and L. Ciminiera, "Radix-8 division with over-redundant digit set," *J. VLSI Signal Process.*, vol. 7, no. 3, pp. 259-270, May 1994.
- [41] N. Quach and M. Flynn, "A radix-64 floating-point divider," Comput. Syst. Lab., Stanford Univ., Stanford, CA, USA, Tech. Rep. CSL-TR-92-529, Jun. 1992.
- [42] H. R. Srinivas and K. K. Parhi, "A fast radix-4 division algorithm and its architecture," *IEEE Trans. Comput.*, vol. 44, no. 6, pp. 826-831, Jun. 1995.
- [43] G. S. Taylor, "Radix 16 SRT dividers with overlapped quotient selection stages," in *Proc. 7th IEEE Symp. Comput. Arithmetic*, Jun. 1985, pp. 64-71.
- [44] T. E. Williams and M. A. Horowitz, "A zero-overhead self-timed 160ns 54-b CMOS divider," *IEEE J. Solid-State Circuits*, vol. 26, no. 11, pp. 1651-1661, Nov. 1991.
- [45] T. Lang and A. Nannarelli, "A radix-10 digit-recurrence division unit: Algorithm and architecture," *IEEE Trans. Comput.*, vol. 56, no. 6, pp. 727-739, Jun. 2007.
- [46] D. Das Sarma and D. W. Matula, "Faithful bipartite ROM reciprocal tables," in *Proc. 12th Symp. Comput. Arithmetic*, Jul. 1995, pp. 12-25.
- [47] M. P. Vestias and H. C. Neto, "Revisiting the Newton-Raphson iterative method for decimal division," in *Proc. 21st Int. Conf. Field Program. Log. Appl.*, Sep. 2011, pp. 138-143.
- [48] P. Saha, D. Kumar, P. Bhattacharyya, and A. Dandapat, "Vedic division methodology for high-speed very large scale integration applications," *J. Eng.*, vol. 2014, no. 2, pp. 51-59, Feb. 2014.
- [49] P. Bannon and J. Keller, "Internal architecture of Alpha 21164 microprocessor," in *Dig. Papers OMPCON Technol. Inf. Superhighway*, vol. 95, Mar. 1995, pp. 79-87.
- [50] S. E. Richardson, "Exploiting trivial and redundant computation," in *Proc. IEEE 11th Symp. Comput. Arithmetic*, Jul. 1993, pp. 220-227.
- [51] J. Cortadella and T. Lang, "High-radix division and square-root with speculation," *IEEE Trans. Comput.*, vol. 43, no. 8, pp. 919-931, Aug. 1994.
- [52] K. Huang and Y. Chen, "Improving performance of floating point division on GPU and MIC," in *Proc. 15th Int. Conf. Algorithms Archit. Parallel Process.*, Zhangjiajie, China, 2015, pp. 691-703, doi: 10.1007/978-3-319-27122-4_48.
- [53] X. Fang and M. Leeser, "Vendor agnostic, high performance, double precision floating point division for FPGAs," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2013, pp. 1-5, doi: 10.1109/HPEC.2013.6670335.
- [54] W. Liu and A. Nannarelli, "Power dissipation challenges in multi-core floating-point units," in *Proc. ASAP-21st IEEE Int. Conf. Appl.-Specific Syst., Archit. Processors*, Jul. 2010, pp. 257-264, doi: 10.1109/ASAP.2010.5540986.
- [55] A. Thall, "Extended-precision floating-point numbers for GPU computation," in *Proc. Special Interest Group Comput. Graph. Interact. Techn. Conf.*, Boston MA, USA, Jul. 2006, p. 52.
- [56] M. Qasaimeh, K. Denolfy, J. Loy, K. Vissersy, J. Zambreno, and P. H. Jones, "Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels," in *Proc. Int. Conf. Embedded Softw. Syst. (ICCESS)*, Jun. 2019, pp. 1-8.

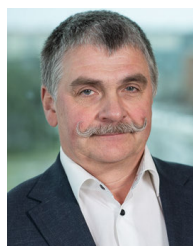
- [57] K. Jun, "Modified non-restoring division algorithm with improved delay profile," M.S. thesis, Fac. Graduate, School Univ. Texas Austin, Austin, TX, USA, 2011.
- [58] S. F. Oberman and M. J. Flynn, "An analysis of division algorithms and implementations," Comput. Syst. Lab., Dept. Elect. Eng. Comput. Sci., Stanford Univ., Stanford, CA, USA, Tech. Rep. CSL-TR-95-675, Jul. 1995.
- [59] J.-S. Chiang, H.-D. Chung, and M.-S. Tsai, "Carry-free radix-2 subtractive division algorithm and implementation of the divider," *Tamkang J. Sci. Eng.*, vol. 3, no. 4, pp. 249–255, 2000.
- [60] N. Burgess and T. Williams, "Choices of operand truncation in the SRT division algorithm," *IEEE Trans. Comput.*, vol. 44, no. 7, pp. 933–938, Jul. 1995.
- [61] B. Mehta, J. Talukdar, and S. Gajjar, "High speed SRT divider for intelligent embedded system," in *Proc. Int. Conf. Soft Comput. Eng. Appl. (icSoftComp)*, Dec. 2017, pp. 1–5.
- [62] D. M. Russinoff, "Computation and formal verification of SRT quotient and square root digit selection tables," *IEEE Trans. Comput.*, vol. 62, no. 5, pp. 900–913, May 2013.
- [63] W. Liu and A. Nannarelli, "Power efficient division and square root unit," *IEEE Trans. Comput.*, vol. 61, no. 8, pp. 1059–1070, Aug. 2012.
- [64] A. Nannarelli, "Performance/power space exploration for binary64 division units," *IEEE Trans. Comput.*, vol. 65, no. 5, pp. 1671–1677, May 2016.
- [65] R. E. Bryant, "Bit-level analysis of an SRT divider circuit," in *Proc. 33rd Design Automat. Conf.*, Las Vegas, NV, USA, 1996, pp. 661–665.
- [66] S. F. Oberman and M. J. Flynn, "Minimizing the complexity of SRT tables," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 6, no. 1, pp. 141–149, Mar. 1998.
- [67] T. M. Carter and J. E. Robertson, "Radix-16 signed-digit division," *IEEE Trans. Comput.*, vol. 39, no. 12, pp. 1424–1433, Dec. 1990.
- [68] D. E. Atkins, "Higher-radix division using estimates of the divisor and partial remainders," *IEEE Trans. Comput.*, vol. C-17, no. 10, pp. 925–934, Oct. 1968.
- [69] R. Trummer, P. Zinterhof, and R. Trobec, "A high-performance data-dependent hardware divider," in *Systems and Simulation, Parallel Numerics*. Ljubljana, Slovenia: Salzburg Univ.; Ljubljana Jožef Stefan Institute, 2005, ch. 7, pp. 193–206.
- [70] R. Erra, "Implementation of a hardware algorithm for integer division," M.S. thesis, Elect. Eng., Fac. Graduate College Oklahoma State Univ., Payne County, OK, USA, Aug. 2019.
- [71] I. Rust and T. G. Noll, "A digit-set-interleaved radix-8 division/square root kernel for double-precision floating point," in *Proc. Int. Symp. Syst. Chip*, Tampere, Finland, Sep. 2010, pp. 150–153, doi: 10.1109/ISSOC.2010.5625547.
- [72] S. Knowles, "Arithmetic processor design for the T9000 transputer," *Proc. SPIE*, vol. 1566, pp. 230–243, Dec. 1991.
- [73] A. Pineiro, J. D. Bruguera, F. Lamberti, and P. Montuschi, "A radix-2 digit-by-digit architecture for cube root," *IEEE Trans. Comput.*, vol. 57, no. 4, pp. 562–566, Apr. 2008.
- [74] N. Takagi, S. Kadowaki, and K. Takagi, "A hardware algorithm for integer division," in *Proc. 17th IEEE Symp. Comput. Arithmetic*, Jun. 2005, pp. 140–146.
- [75] B. R. Lee and N. Burgess, "Improved small multiplier based multiplication, squaring and division," in *Proc. 11th Annu. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2003, pp. 91–97.
- [76] A. Nannarelli and T. Lang, "Low-power divider," *IEEE Trans. Comput.*, vol. 48, no. 1, pp. 2–14, Jan. 1999.
- [77] A. Nannarelli, "Radix-16 combined division and square root unit," in *Proc. 20th IEEE Symp. Comput. Arithmetic*, Jul. 2011, pp. 169–176, doi: 10.1109/ARITH.2011.30.
- [78] H. P. Sharangpani and M. L. Barton, "Statistical analysis of floating-point flaw in the Pentium processor (1994)," Intel Corp., Santa Clara, CA, USA, Tech. Rep., 1994, pp. 1–32.
- [79] E. M. Clarke, S. M. German, and X. Zhao, "Verifying the SRT division algorithm using theorem proving techniques," in *Computer Aided Verification* (Lecture Notes in Computer Science), vol. 1102, R. Alur and T. A. Henzinger, Eds. Berlin, Germany: Springer, 1996, pp. 111–122, doi: 10.1007/3-540-61474-5_62.
- [80] E. M. Schwarz and M. J. Flynn, "Using a floating-point multiplier's internals for high-radix division and square root," Dept. Elect. Eng. Comput. Sci., Comput. Syst. Lab., Stanford Univ., Stanford, CA, USA, Tech. Rep. CSL-TR-93-554, Jan. 1993.
- [81] A. Vazquez, E. Antelo, and P. Montuschi, "A radix-10 SRT divider based on alternative BCD codings," in *Proc. 25th Int. Conf. Comput. Design*, Lake Tahoe, CA, USA, Oct. 2007, pp. 280–287, doi: 10.1109/ICCD.2007.4601914.
- [82] L. Chen, F. Lombardi, P. Montuschi, J. Han, and W. Liu, "Design of approximate high-radix dividers by inexact binary signed-digit addition," in *Proc. Great Lakes Symp. VLSI*, May 2017, pp. 293–298, doi: 10.1145/3060403.3060404.
- [83] J.-A. Pineiro, M. D. Ercegovac, and J. D. Bruguera, "High-radix iterative algorithm for powering computation," in *Proc. 16th IEEE Symp. Comput. Arithmetic*, Santiago de Compostela, Spain, Jun. 2003, pp. 204–211.
- [84] A. F. Tenca and M. D. Ercegovac, "On the design of high-radix on-line division for long precision," in *Proc. 14th IEEE Symp. Comput. Arithmetic*, Adelaide, SA, Australia, Apr. 1999, pp. 44–51.
- [85] H. Nikmehr, B. Phillips, and C.-C. Lim, "Fast decimal floating-point division," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 9, pp. 951–961, Sep. 2006.
- [86] M. D. Ercegovac and R. McIlhenny, "Design and FPGA implementation of radix-10 algorithm for division with limited precision primitives," in *Proc. Conf. Rec. 42nd Asilomar Conf. Signals, Syst. Comput.*, Pacific Grove, CA, USA, Oct. 2008, pp. 762–766.
- [87] M. D. Ercegovac and R. McIlhenny, "Design and FPGA implementation of radix-10 combined division/square root algorithm with limited precision primitives," in *Proc. Conf. Rec. Forty 4th Asilomar Conf. Signals, Syst. Comput.*, Pacific Grove, CA, USA, Nov. 2010, pp. 87–91.
- [88] M. D. Ercegovac and J. M. Muller, "Complex square root with operand prescaling," in *Proc. 15th IEEE Int. Conf. Appl.-Specific Syst., Archit. Processors*, Sep. 2004, pp. 1–11.
- [89] M. D. Ercegovac and J. M. Muller, "Complex division with prescaling of operands," in *Proc. Appl.-Specific Syst., Archit., Processors*, Jun. 2003, pp. 304–314.
- [90] M. Baesler, S. O. Voigt, and T. Teufel, "FPGA implementations of radix-10 digit recurrence fixed-point and floating-point dividers," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs*, Dec. 2011, pp. 13–19.
- [91] M. D. Ercegovac and J. M. Muller, "Variable radix real and complex digit-recurrence division," in *Proc. 16th Int. Conf. Appl.-Specific Syst., Archit., Processors*, Jul. 2005, pp. 316–321.
- [92] D. Wang, M. D. Ercegovac, and N. Zheng, "Design and analysis of high radix complex dividers," in *Proc. 2nd Int. Conf. Comput. Eng. Technol.*, vol. 1, Apr. 2010, pp. V1-84–V1-88.
- [93] M. D. Ercegovac, T. Lang, and P. Montuschi, "Very-high radix division with prescaling and selection by rounding," *IEEE Trans. Comput.*, vol. 43, no. 8, pp. 909–918, Aug. 1994.
- [94] J. D. Bruguera, "Radix-64 floating-point divider," in *Proc. IEEE 25th Symp. Comput. Arithmetic (ARITH)*, Jun. 2018, pp. 84–91.
- [95] N. Burgess, "A fast division algorithm for VLSI," in *Proc. IEEE Int. Conf. Comput. Design, VLSI Comput. Processors*, Cambridge, MA, USA, Oct. 1991, pp. 560–563.
- [96] C. Tung, "A division algorithm for signed-digit arithmetic," *IEEE Trans. Comput.*, vol. C-17, no. 9, pp. 887–889, Sep. 1968.
- [97] L. A. Montalvo, K. K. Parhi, and A. Guyot, "New Svoboda-Tung division," *IEEE Trans. Comput.*, vol. 47, no. 9, pp. 1014–1020, Sep. 1998.
- [98] J.-S. Chiang and M.-S. Tsai, "A radix-4 new Svoboda-Tung divider with constant timing complexity for prescaling," *J. VLSI Signal Process.*, vol. 33, pp. 117–124, Jan. 2003.
- [99] M. Kuhlmann and K. K. Parhi, "Fast low-power shared division and square-root architecture," in *Proc. Int. Conf. Comput. Design. VLSI Comput. Processors*, Oct. 1998, pp. 128–135.
- [100] L. Montalvo and A. Guyot, "Svoboda-Tung division with no compensation," in *Proc. IEEE Int. Conf. VLSI Design*, Jan. 1995, pp. 381–385.
- [101] M. Joldes, O. Marty, J.-M. Muller, and V. Popescu, "Arithmetic algorithms for extended precision using floating-point expansions," *IEEE Trans. Comput.*, vol. 65, no. 4, pp. 1197–1210, Apr. 2016.
- [102] T. J. Kwon, J. Sondeen, and J. Draper, "Floating-point division and square root using a Taylor-series expansion algorithm," in *Proc. 50th Midwest Symp. Circuits Syst.*, Montreal, QC, Canada, Aug. 2007, pp. 305–308, doi: 10.1109/MWSCAS.2007.4488594.

- [103] A. Kumar and T. N. Sasamal, "Design of divider using Taylor series in QCA," *Energy Procedia*, vol. 117, pp. 818–825, Jun. 2017.
- [104] A. A. Liddicoat and M. J. Flynn, "High-performance floating-point divide," in *Proc. Euromicro Symp. Digit. Syst. Design*, Warsaw, Poland, Sep. 2001, pp. 354–361.
- [105] B. Liebig and A. Koch, "Low-latency double-precision floating-point division for FPGAs," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Shanghai, China, 2014, pp. 107–114.
- [106] K. N. Han, A. F. Tenca, and D. Tran, "High-speed floating-point divider with the reduced area," *Proc. SPIE*, vol. 7444, Sep. 2009, Art. no. 744400, doi: [10.1117/12.827850](https://doi.org/10.1117/12.827850).
- [107] J.-A. Pineiro and J. D. Bruguera, "High-speed double-precision computation of reciprocal, division, square root, and inverse square root," *IEEE Trans. Comput.*, vol. 51, no. 12, pp. 1377–1388, Dec. 2002.
- [108] I. Kong and E. E. Swartzlander, "A goldschmidt division method with faster than quadratic convergence," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 4, pp. 696–700, Apr. 2011.
- [109] R. E. Goldschmidt, "Applications of division by convergence," M.S. thesis, Dept. Elect. Eng., Massachusetts Inst. Technol., Cambridge, MA, USA, Jun. 1964.
- [110] B. Pasca, "Correctly rounded floating-point division for DSP-enabled FPGAs," in *Proc. 22nd Int. Conf. Field Program. Log. Appl. (FPL)*, Oslo, Norway, Aug. 2012, pp. 249–254.
- [111] H. F. Ugurdag, F. D. Dinechin, Y. S. Gener, S. Goren, and L.-S. Didier, "Hardware division by small integer constants," *IEEE Trans. Comput.*, vol. 66, no. 12, pp. 2097–2110, Dec. 2017.
- [112] N. Emmart and C. Weems, "Asymptotic optimality of parallel short division," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2016, pp. 864–872.
- [113] N. Emmart and C. Weems, "Parallel multiple precision division by a single precision divisor," in *Proc. 18th Int. Conf. High-Perform. Comput.*, Dec. 2011, pp. 1–9, doi: [10.1109/HiPC.2011.6152712](https://doi.org/10.1109/HiPC.2011.6152712).
- [114] T. Jebelean, "An algorithm for exact division," *J. Symbolic Comput.*, vol. 15, no. 2, pp. 169–180, Feb. 1993.
- [115] *DK Design Suite User Guide, DK Version 4*, document UM-2005-4.2, Celoxica Limited, 2005.
- [116] *Handel-C Language Reference Manual, DK Version 4*, document RM-1003-4.2, Celoxica Limited, 2005.



UDAYAN S. PATANKAR (Member, IEEE) was born in Nagpur, Maharashtra, India, in September 1987. He received the Diploma degree in electronics and communication from the Maharashtra State Board of Technical Education, Mumbai, India, in 2008, and the B.E. degree in electronics design technology and the M.E. degree in electronics engineering from RTMNU, Nagpur University, Nagpur, in 2011 and 2014, respectively. He is currently pursuing the Ph.D. degree with the

Thomas Johann Seebeck Department of Electronics, Tallinn University of Technology, Estonia. He is one of the authors of the book titled *Elements of Vedic Mathematics* (Tallinn Press, 2018). His research interests include mathematics, semiconductor electronics, circuit design, analog-digital circuits, and semiconductor devices. He is also a member of the IEEE Consumer Electronics Society and the Electron Devices Society.



ANTS KOEL (Member, IEEE) was born in Tallinn, Estonia, in August 1962. He received the Diploma degree in industrial electronics from the Tallinn Polytechnic Institute, Estonia, in 1985, the master's degree in 1998, and the Ph.D. degree from the Tallinn University of Technology, Tallinn, Estonia, in 2014. He became a member of the Wessex Institute International Advisory Committee on Materials Characterization and the Chairman of the Steering Committee of the IEEE-Sponsored Baltic

Electronics Conference 2020 organized by TUT.

...