TAL TECH

DOCTORAL THESIS

Area Efficient Design and Implementation of a Novel Divider Circuit Block

Udayan Sunil Patankar

TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY
TALLINN 2025

TALLINN UNIVERSITY OF TECHNOLOGY
DOCTORAL THESIS
29/2025

Area Efficient Design and Implementation of a Novel Divider Circuit Block

UDAYAN SUNIL PATANKAR

TAL TECH PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Thomas Johann Seebeck Department of Electronics
This dissertation was accepted for the defense of the degree 25/04/2025

Supervisor: Dr. Ants Koel
Thomas Johann Seebeck Department of Electronics
Tallinn University of Technology
Tallinn, Estonia

Co-supervisor: Dr. Tamás Pardy
Thomas Johann Seebeck Department of Electronics
Tallinn University of Technology
Tallinn, Estonia

Expert reviewer: Prof. Emeritus Toomas Rang
Thomas Johann Seebeck Department of Electronics
Tallinn University of Technology
Tallinn, Estonia

Opponents: Prof. Dr. Serge Dos Santos, Associate Professor (Hab. Dir. Rech.)
INSA Centre Val de Loire, Blois Campus Department of Industrial Systems
Inserm U1253 iBraiN- University of Tours
Tours, France

Prof. Dr. András Poppe
Department of Electron Devices
Budapest University of Technology and Economics (BME)
Budapest, Hungary

Defense of the thesis: 28/05/2025, Tallinn

Declaration:
Hereby, I declare that this doctoral thesis is my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology, has not been submitted for a doctoral or equivalent academic degree.

Udayan Sunil Patankar _____

signature

European Union
European Regional
Development Fund

Investing
in your future

ISSN 2585-6898 (publication)
ISSN 2585-6901 (PDF)

TALLINNA TEHNIKAÜLIKOOL
DOKTORITÖÖ
29/2025

Uudne efektiivne jagamistehte riistvaraline realisatsioon

UDAYAN SUNIL PATANKAR

TAL TECH
KIRJASTUS

Contents

List of publications

The list of the author's publications, upon which the thesis has been prepared:

I **Patankar, Udayan**.; Koel, Ants, "Review of Basic Classes of Dividers Based on Division Algorithm" in IEEE Access, Vol. 9, 23035−23069. DOI: 10.1109/ACCESS.2021. 3055735.

II **Patankar, Udayan**; Koel, Ants; Patankar, Sunil; Flores, Miguel, "Area Efficient Hexadecimal Divider Circuit Implementation Based on USP-Awadhoot Division Algorithm" in 2021 IEEE International Conference on Engineering, Technology, and Innovation (ICE/ITMC), 2021, pp. 1–8. DOI: 10.1109/ICE/ITMC52061.2021.9570263.

III **Patankar, Udayan**; Koel, Ants; Patankar, Sunil; Flores, Miguel, "Division Method And Circuit" in PTC the International Patent System, International Bureau of the World Intellectual Property Organization, application no.: PCT/IB 2021/054942, submission no.: 054942, Date: 06 June 2021; published on 15-12-2022, publication no WO2022259009. https://patentscope.wipo.int

IV **Patankar, Udayan;** Koel, Ants; Patankar, Sunil; Flores, Miguel, "Novel Data Dependent Divider Circuit Block Implementation for Complex Division and Area Critical Applications," in NATURE Scientific Reports. Sci Rep 13, 3027 (2023). https://doi.org/10.1038/s41598-023-28343-3.

Author's contributions to the publications

The author's contributions to the papers in this thesis were:

I During the research, the author aimed to establish common study points for comparing the various possible division circuit implementations and application possibilities. After surveying existing solutions, the author analyzed and evaluated their conversion logic, critical procedures, constraints, and implementation statistics to determine the optimal option for a given application. In addition, he proposed a new division algorithm development goal to implement dynamic separate scaling operations for input operands to save space overhead and eliminate overlapping conversion logic sections. Furthermore, he drafted the entire manuscript with input and feedback from co-authors and supervisors.

II The author proposed a hexadecimal number system for developing a divider circuit based on a novel USP Awadhoot digit recurrence division algorithm to improve the implementation area. He simulated the VHDL code for the proposed divider circuit using Xilix's VIVADO and Intel Quartus Prime Lite design tools. He also developed a table of all possible input combinations to validate its simulation and hardware implementation. Furthermore, he drafted the entire manuscript with co-authors' feedback, and published the article under the guidance of supervisors.

III The author proposed a divider circuit implementation based on a novel USP-Awadhoot division algorithm. He designed the finite state machine steps for the proposed divider implementation and developed the Awadhoot matrix computation scheme, simulating the VHDL code for hardware implementation. He also drafted the complete manuscript with co-authors' feedback, and published the article under the guidance of supervisors.

IV This article is the extended version of papers II and III. The author presented mathematical models for the proposed novel USP-Awadhoot digit recurrence division algorithm and Baudhayan-Pythagoras triplet algorithm. He proposed the 8-bit, 16-bit, 24-bit, and 31-bit implementations of the novel USP-Awadhoot digit recurrence division algorithm-based divider circuit. He conducted experiments to design, simulate, and verify the VHDL code for the proposed 8-bit, 16-bit, 24-bit, and 31-bit implementations. He designed the truth tables for verifying their performance and validated the results after simulations and hardware implementations. He also drafted the complete manuscript, with co-authors' feedback, and published the article under the guidance of supervisors.

Abbreviations and terms

ALM	Arithmetic and Logical Module
AI	Artificial Intelligence
ASP	Analog Signal Processing
AGC	Automatic Gain Control
AQ	Additional Quotient
BN	Boron Nitride
C_D_d	Dividend complex number
C_D_r	Divisor complex number
cd_enable	Complex enable
C_D_{d1}	USP-Awadhoot Dividend complex number
C_D_{r1}	USP-Awadhoot Divisor complex number
CLB	Configurable Logic Blocks
CLK	Clock signal
CMOS	Complementary Metal Oxide Semiconductor
CPI	Cycles per Instruction
CPU	Central Processing Unit
C_Q	The final quotient complex number
C_Q_i	Imaginary number coefficient of USP-Awadhoot quotient
C_Q_r	Real number coefficient of USP-Awadhoot quotient
C_Rem	The final remainder complex number
C_Rem_i	Imaginary number coefficient of USP-Awadhoot remainder
C_Rem_r	The real number coefficient of USP-Awadhoot remainder
D_d	The dividend
D_r	The divisor
DRC	Digit Recurrence Class
DSP	Digital Signal Processing
Ec	Conduction band energy level
Ev	Valence band energy level
e_r	A small positive fractional value
FD	Flag Digit
F_d_enable	Enable signal for ASP Awadhoot Divider
FIC	Functional Iteration Class
FPGA	Field Programmable Gate Array
GaN	Gallium Nitride
GaSe	Gallium Selenide
GDA	Goldschmidt Division Algorithm
GD_{dn}	Group Dividend
GPU	Graphic Processing Unit
GQ	The final value of the group quotients
GQ_n	Individual group quotient
GQ_{n-1}	Previous iteration group quotient
G_rD_{dn}	Gross Dividend
GSA	Generalized Svoboda algorithm
GSchA	Goldschmidt Algorithm

GSEFIC	Goldschmidt Series Expansion Type Functional Iteration Class
HAC	Hardware Architecture Class
HT	High Temperature
IC	Integrated Circuit
ICT	Information and Communication Technologies
IoT	Internet of Things
I-V	Current - Voltage
LTC	Look-Up Table Class
Mat_Term1	The first triplet matrix term
Mat_Term2	The second triplet matrix term
MD_r	Modified Divisor
MIC	Many Integrated Cores
MSB	Most Significant Bit
ND_{dn}	Net Dividend
ND_r	New Divisor
NRA	Newton–Raphson Algorithm
NRDRC	Non-Restoring Type Digit Recurrence Class
NRFIC	Newton–Raphson Type Functional Iteration Class
NSTA	New Svoboda-Tung Algorithm
NSTDRC	New Svoboda-Tung Type Digit Recurrence Class
NZC	Number of Zeros Cancelled
OFDM	Orthogonal Frequency Division Multiplexing
ρ	Redundancy factor
PLA	Programmable Logic Array
PQ_n	Partial Quotient
Q	The Quotient
q_j	The quotient bit from the j^{th} iteration
Q_Result	Quotient signal
QST	Quotient Selection Look-up Table
R	Residue
r_1	Complex number one is termed as $x_1 + y_1 i$
r_2	Complex number two is termed as $x_2 + y_2 i$
R_{AQ}	The remainder generated during the calculation of the additional quotient
RDRC	Restoring Type Digit Recurrence Class
Rem_Residue	Remainder signal
R_j	The partial remainder of the j^{th} iteration
RNG	Random Number Generator
R_n	Present Remainder
R_{n-1}	Previous Remainder
RST	Reset signal
SBD	Signed binary digit
SDRC	Svoboda Type Digit Recurrence Class
SEA	Series Expansion Algorithm
Si	Silicon
SiC	Silicon Carbide

SOC	System On Chip
SOI	Silicon on Insulator
SRT	Sweeney-Robertson-Tocher
SRTDRC	SRT Type Digit Recurrence Class
STA	Svoboda-Tung Algorithm
STDRC	Svoboda-Tung Type Digit Recurrence Class
SVD	Singular value decomposition
T_Term	The triplet term
TP_Term1	The first triplet product term
TP_Term2	The second triplet product term
TP_Term3	The third triplet product term
TP_Term4	The fourth triplet product term
TSEA	Taylor Series Algorithm
TSEFIC	Taylor Series Expansion Type Functional Iteration Class
xD_d	Real number coefficient of Dividend complex number
xD_r	Real number coefficient of Divisor complex number
yD_d	Imaginary number coefficient of Dividend complex number
yD_r	Imaginary number coefficient of Divisor complex number
Valid_O/P	Computation Completion Acknowledgment
VHRDRC	Very High Radix Digit Recurrence Class
VLC	Variable Latency Class
VLSI	Very Large Scale Integration

List of figures

List of tables

1 Introduction

Enhancement in the semiconductor industry enables the development of new areas of work and studies in the fields of signal processing, statistical data analysis, computational processing, image processing, artificial intelligence, high-performance graphics rendering systems (such as graphic processing units (GPU)), complex systems on chips, central processing units, biomedical equipment, fuzzy control, and space engineering [1-14]. Signal and image processing environments utilize theoretical and applied mathematics for algorithms and hardware that transform preliminary signals from natural and artificial sources into constructive data, which is valuable for application-specific purposes. Figure 1 illustrates the generalized block view of a sensor package. The sensor package comprises a primary sensing element, input channel interface, data conditioning/signal processing, signal/data transmission, storage, reference clock, and power subsystem. The primary sensing element converts a physical quantity into an electrical signal, generating the required information/data component and noise. It requires further signal processing and noise cancellation to improve the quality of sensor signals and extract the relevant properties (such as amplitude, frequency or spectral content, phase, or timing information) from the varying electrical signals [14].

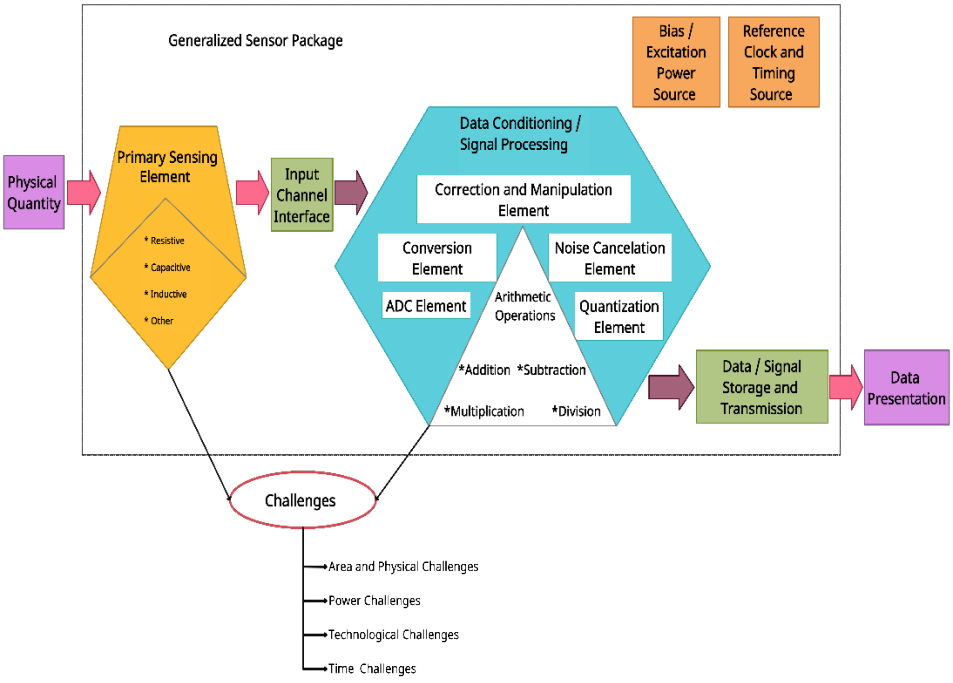


Figure 1. Basic block diagram of the sensor package.

Analog filters have several significant disadvantages that affect filter performance, such as component aging, temperature drift, and component tolerance, with a significant drawback in the inflexibility of the system response. On the contrary, digital signal processing (DSP) is adaptive and flexible, with a high tolerance for component aging and temperature drift. To achieve good results, the DSP system must implement all

mathematical operators in a thoroughly optimized way [1]. The evaluation of addition and multiplication implementations typically falls into the latency range of 1 to 10 clock cycles. The performance evaluation of division operation implementation typically falls into the latency range of 10 to 100 clock cycles [15-19], which is also referred to as 'execution time'. When executing division on *n*-bit operands, recursive subtraction is required for *n* iterations to get the *n*-bit quotient.

The division operation can be replaced by several methods using iterative approaches, such as sequential subtraction (numerical iteration applications) and multiplication (functional iteration applications). Execution time and implementation area are the two basic parameters of comparison. Existing Dividers can be classified into subtractive iteration or functional iteration dividers. Digit recurrence dividers were the first to use successive subtractions, beginning with the least significant bit to calculate the required quotient. For *n*-bit operands, the division requires *n* recursive subtraction iterations to produce the *n*-bit quotient. Digit recurrence dividers are easy to implement for larger bit size operands, due to subtractive iterations, but require extensive conversion time, chip implementation area, and a critical selection logic and overlapping region for quotient bit selection. The SRT (Sweeney-Robertson-Tocher) divider, named after the researchers, is one of the most implemented non-restoring digit recurrence type dividers. SRT divider, also known as the radix-*n* divider. The radix number (*n*) determines how many quotient bits are calculated in a single iteration. In SRT dividers, the radix size is typically kept small because increasing the radix not only increases the number of quotient bits generated per iteration but also significantly complicates the quotient bit selection logic. This complexity arises from the need to handle a larger set of quotient values and manage overlapping regions [15, 20-21]. High-radix division algorithms are implemented with different architectures (e.g., the array structure or cascading architecture) but require a comparatively higher chip implementation area [15]. Overflow, due to overcompensation, causes the selection of a quotient digit out of the range [22-25] and is one of the possible drawbacks of the Svoboda and Svoboda-Tung algorithm-based radix-n divider, that only requires a few Most Significant Bits (MSBs) of the partial remainder for the quotient selection logic.

Functional iterations compute the quotient bit based on the estimation or approximation of series expansion functions, such as the Newton-Rapson [26-27], Goldschmidt [11, 28-31], and Taylor series [11, 32-35]. These require the selection of a reciprocal value at the initial iteration of the conversion. It makes the quotient bit selection logic critical and complex. Nevertheless, the precision of the outcome and the possibility of error are contingent on the proximity of the initial reciprocal selection and rounding off the approximate solution values, rather than infinitely precise ones. The error depends on the accuracy of the initial estimation. Reducing the error requires introducing a trade-off between the additional chip area for the look-up table and the latency of the divider. The Goldschmidt algorithm is a second functional iterative divider that is only effective for floating-point division because it does not provide the remainder [31]. Taylor series expansion calculates an accurate anti-divisor (reciprocal) to reduce the error in the least significant bits of quotient precision with a parallel powering section, causing extra hardware overhead. The upcoming application areas of high-speed computation, embedded systems, artificial intelligence [3, 7-8, 36-37], complex SOC [9], vision systems [1, 5-6, 36-37], automotive control [9], telecommunications [36-37], the internet of things (IoT), cryptography [4, 36-37], and many others, offer the possibility and requirement of further improvements in division implementation.

However, a research gap exists for simultaneously utilizing multiple performance improvement techniques with individual input operands. This provides the possibility of developing a new technique or combination of a fast or moderate, less complex quotient selection logic and methods to reduce the chip implementation area. Reducing the implementation area of the divider block can enhance the overall resource utilization of the larger system in which it is integrated.

The work presented in this thesis contributes to the significant challenges of quotient selection logic criticality, the chip implementation area reduction of hardware, and circuit implementation for the divider, by proposing a novel division algorithm. The ultimate target of the research is to provide simple quotient selection logic, with dynamic separate scaling operations for the dividend and divisor, to reduce the possibility of devastating and costly problems (causing system failure) and reducing the implementation area for digit recurrence divider implementation. The proposed novel divider implementation includes a detailed step-by-step conceptualization, outlining the implementation requirements, and provides resource utilization statistics for further study and implementation with various applications.

1.1 The scope and the organization of the thesis

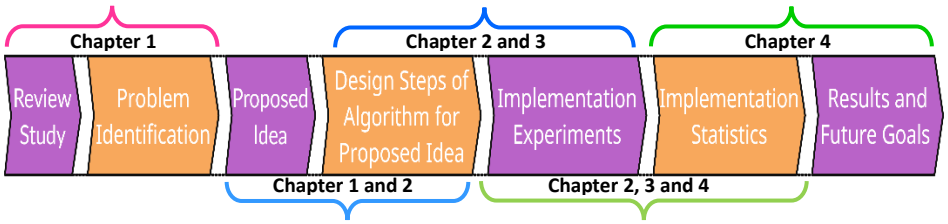


Figure 2. The scope and the organization of the thesis.

Figure 2 illustrates the progression of the concept of a new divider toward a physical implementation and evaluation through its various phases. The scope of the overall thesis is focused on the research required to provide simple quotient selection logic in order to reduce the possibility of devastating and costly problems, which could cause operation failure and reduce the implementation area for digit recurrence divider implementation.

Chapter 1 presents a detailed study of the major concepts, methods, techniques, and algorithms regarding divider implementation, including the working idea, requirements, and implementation statistics. This chapter provides a comprehensive overview of division operation and explains various ways of classifying division operation implementation with detailed information on specifications, advantages, and disadvantages.

Chapter 2 describes the definition and conceptualization of the proposed novel USP-Awadhoot divider implementation.

Chapter 3 presents a detailed description of the application of the novel USP-Awadhoot divider for complex number division using the Baudhayan-Pythagorean triplet algorithm.

Chapter 4 presents a detailed comparative analysis study of the implementation statistics for resource utilization of the proposed novel USP-Awadhoot divider. Waveform analysis explains the idle working state of each signal in the novel USP-Awadhoot divider.

It also describes the road map for future research activities to refine the implementation along with different applications.

Appendix 1 to Appendix 7 present publication details, examples of the proposed novel USP-Awadhoot divider, and functional waveforms.

1.2 Problem statement and research objectives

Division is a derived operation, similar to multiplication. The division operation can be replaced by several methods using iterative approaches, such as sequential subtraction (numerical iteration applications) and multiplication (functional iteration applications). Multiplicative or functional iterative algorithms are faster than subtractive algorithms but require a larger area on a chip for implementation; whereas, subtractive algorithms require less area but have longer execution time. The overlapping region refers to a range of partial remainder values, where the selection of the next partial quotient is ambiguous due to the step size of a radix-n divider. It could cause a problem in selecting the true quotient value. Research and implementation have been carried out for alternative approaches to design a quotient selection logic, which requires only a few MSBs of the partial remainder. Because of this, the final remainder value cannot be calculated at the end of the division. Thus, such a divider is limited to the applications that do not require remainder data.

Many researchers have worked on various performance improvement techniques, such as pre-scaling operands, carry-save remainders, array implementations, truncations, cascading, and differential LUTs. However, these performance improvement techniques have yet to be fully explored to address the research gap of utilizing multiple performance improvement techniques simultaneously with individual input operands. This approach could potentially lead to the development of a new technique or a combination of fast or moderate methods to optimize conversion time and implementation area. Thus, the main objective of the present research is to provide a combination of multiple techniques that can be simultaneously utilized on the individual input operands to achieve an area-efficient solution for divider circuit implementation. This Ph.D. research focuses on the following research objectives (RO).

- RO1 – Investigate the currently existing divider solutions to understand the different concepts of conversion logic, conceptualize the trade-off between area, speed, and power, and propose a suitable option or combination of options to develop an efficient divider.
- RO2 – Develop the theory of conversion logic to implement dynamic separate scaling operations for input operands. Here, a separate scaling operation means simultaneously using different scaling operations for input operands. A partitioning operation is used for the dividend. An operation composed of “Veshtanam Sutra (by osculation) and Lopanasthabhyam sutra (by elimination and retention)” is used for the divisor. ‘Dynamic’ refers to the different values of “Flag Digit (*FD*) and Number of Zeros Cancelled (NZC)”, used in Veshtanam Sutra (by osculation) and Lopanasthabhyam sutra (by elimination and retention), depending on the combination of input operands.
- RO3 – Divider algorithm formulation to reduce the criticality of conversion logic by eliminating overlapping regions in quotient selection.
- RO4 – Implement the divider based on the formulated algorithm, and improve the area requirements to compose the operand-dependent divider circuit design.

1.3 Thesis contribution

The main objective of this doctoral dissertation is to explore the research gap in the simultaneous use of multiple performance enhancement techniques with individual input operands, aiming to design and implement a divider circuit block with reduced area. It also intends to provide a solution, based on the divisor and the dividend relationship, that improves the quotient bit calculation logic and avoids rounding-off errors. As stated before, this Ph.D. research work relate to the derivation of a new algorithm for reduced area implementation of the divider circuit block. The design is developed by simulating the proposed approach and cross-verifying it by performing regular sequential and pseudo-random performance analyses against standard result tables generated by simulations and the theoretical study of the proposed idea. Table 1 summarizes the thesis contributions, in relation to the research papers. The novelty and main contributions of the Ph.D. thesis are as follows:

- In association with RO1, RO2, RO3, and RO4, this thesis contributes to the development of a novel algorithm for implementing a divider circuit block. The innovative concept of dynamic separate scaling operations for the dividend and divisor reduces resource requirements, resulting in a divider circuit block with a low area footprint.
- In association with RO2, RO3, and RO4, I developed an easy Group Quotient (GQ_n) value selection logic in the proposed divider circuit block based on the unique relation derived between Dividend Groups (GD_d), Modified Divisor (MD_r), and Flag Digit (*FD*) without any critical overlapping.
- In association with RO2, RO3, and RO4, I developed a clear process for selecting the final quotient based on the Group Quotient (GQ_n), Partial Quotient (PQ_n), and Additional Quotient (*AQ*) values without critical overlapping regions.
- In association with RO2 and RO3, I implemented a complex divider based on the Baudhayan-Pythagorean triplet algorithm with the proposed USP-Awadhoot divider circuit block.
- The described steps reduce the criticality of the conversion logic by eliminating overlapping regions in the quotient bit selection logic.

Table 1. Publications containing the thesis' contributions.

Contributions	Publication I	Publication II	Publication III	Publication IV
RO1- Review	✓	✓		✓
RO2- Develop dynamic separate scaling operations		✓	✓	✓
RO3- Divider Algorithm formulation			✓	✓
RO4- Divider Implementation and Improvements		✓	✓	✓

2 Division circuit block – overview of division algorithms

This chapter is based on publications I, II, and IV. In the past, limited communication and transportation made it difficult to establish uniform mathematical standards worldwide. Hindu-Arabic numerals, which comprise the ten symbols – 1, 2, 3, 4, 5, 6, 7, 8, 9, and 0, are based on India's decimal number system. They were mentioned in Aryabhata's 'Aryabhatiya' and Brahmagupta's 'Brahmasphuta Siddhanta' in the 6th and 7th centuries, and, according to al-Qifti, they were introduced to the Arab world in the late 7th century. Later, in the 12th century, these were transmitted to Europe via the chronologies of the scholars, particularly al-Khwarizmi, al-Kindi, and the Italian mathematician Leonardo Pisano (also known as Fibonacci) [38-41].

Although new concepts, operations, logic, and relations have been developed in mathematics, 'addition', 'subtraction', 'multiplication', and 'division' are still the firm foundations of applied mathematics [1, 10]. Due to the commutative and associative properties of addition and multiplication, operands can be rearranged flexibly without affecting the result [42-43]. The division operation is a derived operation in the same way that multiplication is also but, instead of successive addition, it is derived by successive subtraction, along with some controlling conditions. Similar to subtraction, division also lacks commutative and associative properties, making its implementation in electronic circuitry critical and challenging. Thus, it is essential to understand the importance of the critical parameter requirements and problems associated with the implementation of a division circuit block.

2.1 Importance of division circuit blocks

A Field Programmable Gate Array (FPGA) is an advanced technological feature. It provides hardware re-programmability, which reduces implementation time and hardware costs. It gives the flexibility to implement a system on a chip for different purposes. The Arithmetic and Logical Modules (ALMs) of FPGAs are essential building blocks for implementing desired logic [44]. FPGA applications are more critical for automotive control, online data processing, and a wide range of computational tasks, which could be solved by implementing a small, complex system (such as a computer system) on a single chip. In general-purpose applications, central processing units (CPU/processor) perform division with several iterations, even for a few bits. This problem becomes critical, along with an increasing bit count [10]. Such issues are even more severe in the graphics processing unit (GPU) and Intel's many integrated core (MIC) architecture, which provides parallel architecture [45]. The CPU's working frequency has increased to 3 GHz over time; however, this has also led to higher power dissipation [12].

Complex division used in various applications in essential engineering works, such as earth fault distance protection, acoustic pulse reflectometry, astronomy, non-linear radio frequency measurements [46,53], and control theory applications (e.g. investigating root locus, Nyquist plot, Nichol's plot, and microwave system frequency response) [47]. It is also required in digital signal processing and numerical computation applications, such as Vertical Bell Laboratories layered space-time detection (V-Blast), orthogonal frequency division multiplexing, and channel equalization of the MIMO system [48]. Earlier, the lack of a dedicated divider (due to its low usage and high chip area requirements) resulted in an emphasis on division operations performed by software [6, 18, 49-51]. Designers also have to consider the implementation technology for the

algorithm as it is directly related to the area and time concerns of divider circuit implementation [51-52].

As per the studies presented in [15-19] and [51-52], the typical latency for addition and multiplication ranges from 2 to 8 clock cycles. In contrast, division latency ranges from 8 to 80 clock cycles [53]. The division could be performed using adders and multipliers, instead of creating separate hardware for the divider. Such an arrangement to perform a division operation comes with a significant risk of extended overhead and possible error in the final result due to rounding off. The algorithms and architectures studied in [52, 54] show that the focus was placed on improving adders and multipliers rather than developing a dedicated divider circuit. However, solely prioritizing these improvements can lead to undesirable behavior, including numerical vulnerabilities and a higher risk of overflow. [51, 52, 54].

Even if it were possible to build a multiplier to compute in a single cycle, finding a matching adder would be difficult. Fast-operating multipliers, such as array multipliers, can have low cycles to execute at the cost of significant area overhead, which would not be a cost-effective area solution for implementing division operation [51, 52]. This indicates that the area distribution among the adder, multiplier, and divider circuits should be proportionally balanced to have an efficient and optimized system. As per the study presented in [51, 52], iterative algorithms are preferred over pure combinational algorithms, for implementing division circuit blocks to achieve a low area footprint system. The study also suggested that neglecting improvements in division operation implementation is a key factor contributing to performance loss in embedded systems, digital systems, digital circuits, computer systems, and integrated circuits. Thus, it is necessary to focus on low-area footprint divider circuit implementation, as the implementation area introduces critical delays and timing issues in its standard execution. Also, the increase in the application demand and development of new application areas encourage the development of area-efficient divider circuit blocks.

2.2 Division circuit block taxonomy

A study conducted by [55] demonstrated that the installation of division circuit blocks influences the performance of a complicated system. In addition, even the slightest change, such as a 1% improvement in the performance of the division circuit block, might affect the system's performance by up to 20%. Division operations were performed based on sequential, linear operations and digital circuitry in the applications with low computation requirements, to express logic functions with high accuracy on account of the large area and latency [56-58]. Implementation area, computation time, and power consumption are the three main topics of interest from a system implementation point of view. The applications developed at this stage required area reduction for the division because their current implementations lack area and latency efficiency [59-61]. Area efficiency refers to the percentage of the available hardware resources utilized to implement the divider. An algorithm can be specified as a computer program or a hardware circuit design with specifications that describe the computational procedure to be followed during implementation [62]. Thus, many methods or algorithms have been researched, designed, and implemented over time, with the common goal of an efficient divider circuit implementation for an efficient system.

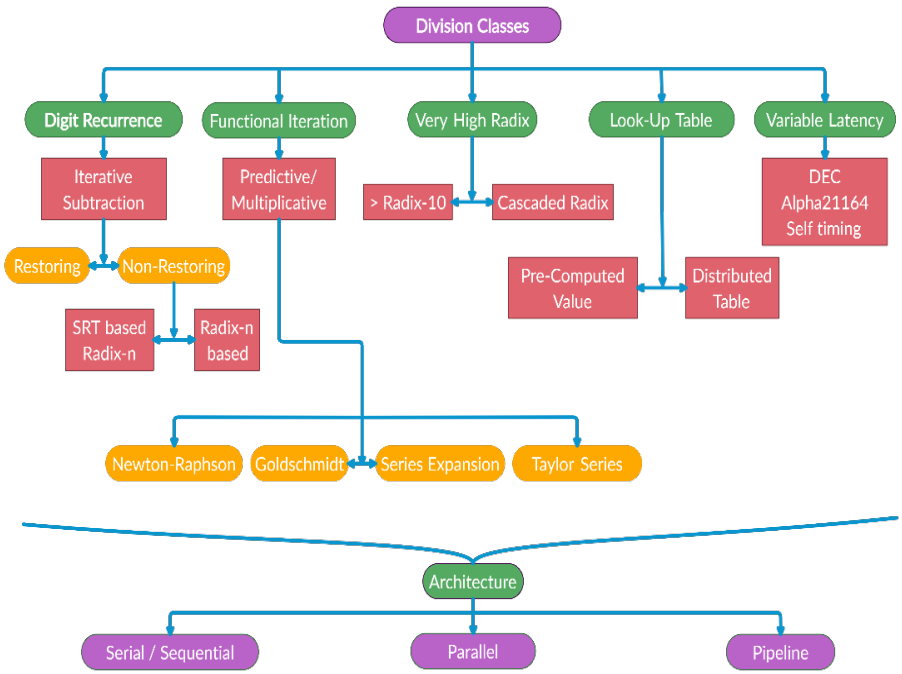


Figure 3. Division algorithm taxonomy.

In the studies described in [7, 15, 31, 63-65, 71-73], numerous mathematical algorithms were devised and analyzed over several decades. Many algorithms are difficult to distinguish precisely, but they may be divided into two categories: digit recursion and functional iteration [65]. Still, depending on the quotient conversion logic, they can be broadly summarised into multiple divider classes. The study presented in [15] showed that division algorithms can be categorized into five distinctive classes. The hierarchical distribution of various classes of division algorithms is expressed as division algorithm taxonomy in Figure 3 and described based on the four factors representing conversion logic, hardware architecture, performance, and execution type [6, 15, 66-67]. There are five broad categories of division algorithms: digit recurrence, functional iteration, very high radix, look-up table, and variable latency. Based on the hardware architecture and access techniques, they can be further classified as serial, sequential, parallel, pipeline, slow, fast, iterative, and predictive.

2.2.1 Digit recurrence class (DRC)

The digit recurrence class (DRC) of the division algorithm is the earliest and most pioneering class among all division algorithms [15, 20]. The quotient is calculated using a series of successive subtraction operations, beginning with the least significant bit [1-3, 15, 20, 66-70]. The digit recurrence class of algorithm-based dividers is categorized into two types of dividers, commonly known as ‘restoring’ and ‘non-restoring’ algorithm-based dividers.

Many processors such as Intel Pentium, HP PA 8000, and Sun UltraSPARC [70] initially implemented a restoring type DRC algorithm-based long division divider concept. Thus, the remainder and quotient values remain either positive or zero [1, 3, 20, 61, 68, 71-73]. The SRT algorithm is one of the most popular non-restoring digit recurrence

division algorithms to implement. Furthermore, many attempts [74-105] have been orchestrated to develop, investigate, and discuss the original concept of the SRT algorithm, in order to improve it. The trade-off between the components of the choices to be made is mainly the radix, quotient, and partial remainder representation [15, 50, 97-98], resulting in diverse application selections ranging from less critical to crucial, which impacts time-cost requirements. A rise in radix value increases the size of the quotient selection logic table, beyond the practical limits of the implementation. It is evident that Intel has lost millions of dollars due to the Pentium processor's flaw in the overlapping region of the floating-point divider [92, 104]. In 1963, Svoboda devised another digit recurrence division algorithm based on the partial remainder alone. It considers quotient digit selection logic based on the remainder's MSBs [8, 22-25, 119-122]. Tung [22-24, 119, 121] investigated the potential of implementing the Svoboda algorithm using a signed digit number system.

2.2.2 Very high radix digit recurrence class (VHRDRC)

Unlike SRT and other radix-n divides, these very high radix algorithms have different hardware and circuitry for quotient selection and partial remainder generation. The high radix algorithm proposed by Wong and Flynn [123] requires at least one look-up table, comprising$\left(2^{(m-1)} \times m\right)$ bits. The high radix algorithm proposed by Lang and Nannarelli [124] shows the construction of a radix-2^k divider to implement a radix-10 divider whose quotient digit is partitioned into two sections, one in radix-5 and the other in radix-2.

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 [15]. The possible methods, which are applicable to high radix dividers, include the use of: different look-up tables for quotient digit selection logic [23, 93, 125], pre-scaling operands [126-131], Fourier division [132-133], alternative digit codes (like BCD digits instead of decimal and basic binary digits [105]), cascading multiple stages of lower radix dividers [45], overlapping two or more phases of low radix [85, 94], a truncated schema of exact cell binary shifted adder array [100, 134-135], on-line serial and pipelined operand division [136], the parallel implementation of low radix dividers [137], and array implementation [4].

2.2.3 Look-up table class (LTC)

Look-up tables can hold the pre-computed values, standard values and exact values of the approximation of the reciprocal for the quotient bit finalizing technique. The latest development described in [138] pertains to the bipartite reciprocal table, which can be utilized for reciprocal approximation in dividers. It uses two separate look-up tables for positive and negative values. The look-up table class is a hybrid class of dividers, as look-up tables are used to improve dividers from different classes.

2.2.4 Functional iteration class (FIC)

This division method uses successive multiplications instead of subtractions. It is possible to get multiple quotient digits in a single iteration but at the cost of accuracy, due to the rounding off of solution values and implementation area [139]. The Newton-Raphson method is used in IBM 360/91 and Astronautics ZS-1 [26-27]. Taylor Series Expansion is used in IBM RS/6000 and AMD K7 processors [33, 35]. Later, J. Liu et al. [33] presented a hybrid algorithm that combined prescaling, series expansion, and Taylor series expansion for the implementation of a divider.

2.2.5 Variable latency class (VLC)

The DEC Alpha 21164 is one of the best examples of a variable latency class algorithm implementation based on the basic normalizing non-restoring division algorithm [15, 50, 145]. Sometimes, multiple stages are cascaded together with a self-timing partial remainder in the self-timing technique [118, 146]. The Hal SPARC V9 processor and Sparc64 are examples of practical implementations of the variable latency self-timing division algorithm [15, 50]. Richardson [147] described a mechanism for caching results that can be used with the divider to accelerate calculations in applications involving repeating operands. In [89], Cortadella mentioned implementing the SRT divider with variable latency, detecting a variable number of quotient bits at each iteration.

2.3 Hardware architectures

The three primary classifications of hardware architecture are: sequential or serial, parallel or concurrent, and pipelined or hybrid. The serial hardware architecture consists of the sequential implementation and processing of the components required for algorithm implementation and is primarily used for general purposes. Subtractive iteration-based digit recurrence division algorithms are the best examples of serial dividers [20, 36, 136, 148]. The parallel hardware architecture consists of multiple hardware units implemented and processed concurrently to get the desired result with fewer iterations. This approach is mainly used for graphical processing units (GPUs) and in Intel's many integrated core (MIC) processors [6, 10, 13, 149-150]. A parallel divider, based on the Jebelean exact division algorithm [149-151], is another example of a parallel hardware architecture class divider. The third approach provides parallel processing by executing the instruction level overlapping of a computational approach [4, 9, 20, 91, 136]. This architecture allows the simultaneous performance of several instructions of the computation process to achieve some degree of parallelism. Pipeline work structure can be achieved by designing a computational logic that provides functional overlap in the execution stage and arranging pipelined hardware, like a fully pipelined array structure [4, 9].

2.4 Performance improvement techniques

Performance-improvement techniques, like simple staging, overlapping/pipelined execution, overlapping quotient selection, overlapping partial remainder computation, range reduction, operand scaling, and circuit family effects are significant in divider implementation. HP PA-7100 [15, 106] and AMD 29050 [15, 107] microprocessors are examples of two radix-4, clocking faster than the system clock to perform radix-16 work in every machine cycle [15, 108]. The AMD 29050 microprocessor also exhibited the same logic of achieving higher radix. The study presented in [85, 109] showed that many circuit-level implementations of the SRT algorithm yield different performances, depending on the choice of circuit family. In the overlapping/pipelined execution, the partial remainder-dependent pipelined form of execution is performed when a redundant format represents the partial remainder. In contrast, the quotient selection execution-dependent pipeline is suitable when a non-redundant format represents the partial remainder [110].

The technique of reducing the divisor by a fixed factor, to bring it as close as possible to one, is known as ‘divisor pre-scaling’ [15, 111]. The basic concept of the pre-scaling divisor and dividend, by common pre-scaling factor, is explained in [112-113]. A similar concept was explained in [102] and it was suggested that the user uses six digits of the

redundant partial remainder to generate quotient bit selection logic in implementing the Radix-4 divider. Performance improvement techniques can also be considered for other classes of division algorithms depending on the particular requirements of individual algorithm class-based dividers [83, 102, 109-118]. However, no single performance improvement technique can concurrently address all performance factors, and one has to decide what type of option to select based on the particular application.

2.5 Summary of comparative analysis

An efficient divider is required for an effective and efficient computation system. Table 2 summarizes a comparative study of the different division algorithm-based dividers. The initial distribution gives digit recurrence, functional iteration, very high radix, a look-up table, variable latency, serial/sequential, parallel, and pipelined classes of a divider [6, 15, 23, 50, 68]. Digit recurrence is the most trusted, implemented, researched, and commercially used division class amongst all divider implementation classes. The restoring, and some non-restoring, algorithms implement simple conversion logic but require a long time and a large area. Functional iterative class dividers compute the quotient bits by estimating or approximating series expansion functions such as, Newton-Rapson [26-27], Goldschmidt [11, 28-31], and Taylor series [11, 33-35], where an approximated reciprocal multiplies the dividend to converge toward the required quotient. They use multiplication instead of subtraction, which decreases the number of iterations and provides several quotient digits with minimal latency in a single iteration.

However, multipliers require a larger footprint than adders or subtractors. Multiplication makes functional iteration dividers more complicated than basic digit recurrence dividers. This divider has the significant drawback of the quotient bit's inaccuracy because of direct rounding off of the approximate solution values, rather than infinitely precise ones. In the Newton-Raphson iteration, which is limited to two multiplications and must proceed in series, a significant error is generated. The generated error depends on the accuracy of the initial estimation. Reducing the error requires introducing a trade-off between the additional chip area for the look-up table and the latency of the divider.

Unlike the Newton-Rapson method, which only multiplies the dividend, the Goldschmidt algorithm multiplies the dividend and the divisor by the anti-divisor. It is only useful for floating-point division because it does not offer the remainder [31]. Another drawback is that 1's complement can avoid carry propagation delay but it adds a new approximation error in each iteration. In Taylor series dividers, series expansion computes an accurate anti-divisor (reciprocal) to reduce the error in the least significant bits of quotient precision, with a parallel powering section that calculates high-order terms, increasing the hardware overhead. Variable latency class [89, 124, 145, 147] dividers are uncommon due to their complexity and large area. High radix [124] reduces the latency but requires a large capacity look-up table, which is impractical for implementation. The look-up table class [67, 138] involves storage like ROM, which increases the area requirements for implementation. Dividers can be implemented using one of three distinct hardware architectures.

The serial hardware architecture [20, 36, 136, 148] necessitates increased latency and conversion time, making it unsuitable for mission-critical applications. In contrast to serial architecture, parallel hardware architecture [6, 10, 13, 149-150] requires the concurrent operation of multiple cores, precise synchronization, and a significant implementation area, resulting in a higher implementation cost.

Table 2. Summary of a comparative study of different dividers.

Sr. No.	Algorithm	Equations	Important Points
1	Restoring Divider [1-3, 15, 20, 61, 66-73].	For J[th] iteration $q_j = 0 \quad if \quad R'_j < 0$ $q_j = 1 \quad if \quad R'_j \geq 0$ $R_j = 2R_{j-1} \quad if\, q_j = 0$ $R_j = R'_j \ \ if\ q_j = 1$ $R'_j = 2R_{j-1} - D_r$	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 not 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 the subtractor, shift register, and multiplier give the approximate area requirement for algorithm implementation. Possible loss of most significant bit (MSB) and checks for overflow are required. Requires full-width comparison in every iteration to get one bit of quotient. The quotient needed to be rearranged to get the actual quotient.
2	Non-Restoring Divider [1-3, 15, 20, 61, 66-73, 174].	For J[th] iteration $q_j = -1 \quad if \quad R_{j-1} < 0$ $q_j = 1 \quad if \quad R_{j-1} \geq 0$ $R_j = 2R_{j-1} + D_r \, if\, q_j = -1$ $R_j = 2R_{j-1} - D_r \, if\, q_j = +1$	Like the restoring algorithm, it does not require the restoring of the partial remainder if subtraction becomes negative. No requirement for a look-up table. Based on the previous iteration sign value of the partial remainder, only one addition or subtraction can be performed in each iteration. Partial remainder kept between -Dr to +Dr and quotient digit is -1 or 1. It requires a sign bit to decide whether to perform addition or subtraction; the adder, subtractor, and shift register give the approximate area requirement for algorithm implementation. Requires an extra bit to be added with the partial remainder, to have a track on a sign. It requires a separate adder and subtractor in each iteration. Area utilization of implementation is approximately equal to the area required to implement an adder, subtractor, and shift register.

3	SRT Divider [15, 50, 66 74-105]	For J[th] iteration $q_j = \overline{1} \quad if \quad 2R_{j-1} < -D_r$ $q_j = 0 \quad if \quad -D_r \leq 2R_{j-1} \leq D_r$ $q_j = 1 \quad if \quad 2R_{j-1} \geq D_r$ Has one of the values *-m, -m+1… -1, 0, +1. . . m-1, m*, where *m* is an integer comprising k digits of radix-n as $\frac{1}{2}(n-1) \leq m \leq n-1$ $n = 2^b \quad and \quad k = {}^{x}/_{b}$ $Q = \sum_{j=1}^{k} q_j n^{-j}$ Quotient *q* is generated as a dividend division by a divisor of *x* most significant bits retiring b bits of the quotient in each iteration. It is called a 'radix-n performing *k* iterations' to get the desired quotient.	It is a non-restoring algorithm based on radix-n.
			Named after Dura W. Sweeney [74], James E. Robertson [75], and Keith D. Tocher [7].
			For *x* bits, integer division requires *k=x/b* iterations, *b* is the number of bits detected in each iteration.
			n decides how many quotient bits are 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, ….., -1, 0, 1, ……., m-1, m*}.
			The algorithm implements 2's complement value of D_r instead of D_r , which provides shifting over zeros to eliminate extra adders and subtractors.
			It needs an extra subtractor to find out the next partial remainder.
			Error results due to few MSBs being used to predict quotient bits as in low radix, which decreases with the increase of radix.
			Quotient select table plus carry-save adder (CSA) gives the approximate area requirement for algorithm implementation. It shows the iteration time of accessing the select quotient table plus multiple forms and subtraction. It requires a quotient selection look-up table.
			Selecting higher quotient bits causes complexity in quotient selection logic, and higher radix implementation is complex due to impractical multiples of the divisor.
			It needs to convert the last remainder to conventional representation to find the sign bit, and the quotient correction stage selection depends on the sign bit.
4	Very high radix [4, 15, 23, 85, 93-94, 100, 123-137]	*******	It retires more than ten quotient bits in one iteration and requires a large look-up table with a bigger capacity for quotient selection logic. A lookup table is required for obtaining an initial approximation to reciprocal and quotient digit selection logic.
			It uses multiplication to form divisor multiples.
			It differs from the regular radix-n divider regarding the number and type of operations used in each iteration and quotient digit selection logic.
			High radix makes quotient selection logic more complex and impractical to implement

5	Taylor Series [11, 33, 35, 144]	$q = {D_d}/{D_r}$ and $X_0 = {1}/{D_r}$ $q = D_d X_0\{1 + (1 - D_r X_0) + (1 - D_r X_0)^2 + (1 - D_r X_0)^3\}$ D_d = Dividend and D_r = Divisor $1/D_r = Antidivisor$	It is a multiplicative iteration-based algorithm, hence requiring a large area.
			The precision depends upon the closeness to the 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.
6	Newton-Raphson [26-27, 139-141]	$Q = D_d/D_r = p \times (q)^{-1}$ $f(X) = 1/X - q^{-1} = 0$ $X_{i+1} = X_i - \frac{f(X_i)}{f'(X_i)}$ $X_{i+1} = X_i - \frac{(1/X_i - q^{-1})}{1/X_i^2} = X_i \times (2 - q^{-1} \times X_i)$ $\epsilon_{i+1} = \epsilon_i^2\ (q^{-1})$ p = Dividend and $(q)^{-1} = Antidivisor$	The accuracy can be improved by selecting a proper root at the beginning.
			Latency and error in convergence are directly dependent on the root selected at the beginning of the convergence and show the iteration time approximately equal to the time required for two serial multiplications.
			Multiplier, quotient select look-up table, and control logic give the approximate area requirement for algorithm implementation.
			The final quotient is derived by multiplying the approximated reciprocal and dividend.
			Shows error due to inaccuracy of quotient digit prediction or estimation.
			It requires multiplication and addition or subtraction at each iteration; using 1's complement includes more error.
7	Goldschmidt [31, 142-143]	$D_d/D_r = N/D = A/B$ $x_{n+1} = x_n(2 - y_n) = x_n r_n$ $y_{n+1} = y_n(2 - y_n) = y_n r_n$	It is a convergence-based functional iterative class divider algorithm.
			It multiplies both dividend and divisor by the 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 give the approximate area requirement for algorithm implementation.
8	Variable Latency [15, 50, 89, 118]	******	Variable conversion times for different sets of dividends and divisors due to the varied rate of quotient bit retiring
			Self-timing, result cache, and speculation of quotient digit are some techniques used to provide variable latency.

8	Variable Latency [145-147]		The DEC Alpha 21164 is one of the best examples of variable latency class algorithm implementation, based on the concepts of the simple normalizing non-restoring division algorithm.
9	Svoboda Algorithm And Svoboda-Tung Algorithm [8, 22-25, 119-122]	$\left\{\frac{mn}{(m+1)(n-1)} < D_r < \frac{m(n-2)}{(n-1)(m-1)}\right\}$ $\{-m/n-1 < R_j < m/n-1\}$ $Range = \{0, \pm 1, \ldots\ldots, \pm m\}$ $Boundry\ limit = \{n/2+1 \leq m \leq n-1\}$ $m = Range\ of\ SBD\ and\ n = Radix$	The quotient digit is predicted based on the partial remainder without considering the divisor; one or two MSBs of the partial remainder are used for generating the quotient digit selection logic.
			It can select a quotient digit out of the radix range as an overflow occurred due to compensation.
			It requires pre-scaled operands and can work on conventional and signed digit ranges.
			It is also a radix-n-based algorithm with sign binary digit numbers, like the SRT algorithm.
			It is applicable to more than radix 4 and pre-scaled operands are required; it needs extra multipliers, resulting in more hardware overhead.
10	Smaller Dividend [6]	$N_1 = \sum_{i=0}^{2n-1} x_{2n+i} 2^{2n+i}$ $N_2 = \sum_{i=0}^{2n-1} x_i 2^i$ $D_d = N_1 + N_2$ $D_d/D_r = (N_1 + N_2)/D_r = N_1/D_r + N_2/D_r$	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., the dividend bit counts as 4n and the divisor bit counts 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.
11	Jebelean Exact Division [149-151].	$D_d = d * Q$ $D_d = D_{dupk} n^k + D_k$ $b_k = (-n^{-k} D_{dk})\ _{mod\ d}$ $b_k = (-n^{-k}_{mod\ d} D_{dk})\ _{mod\ d}$	It applies when a completed division is performed on long integer operands in digital computation, even after knowing that the remainder is zero.
			It works from the least significant digit of the operands.
			Remarkable performance is observed when radix is a prime or a power of 2.
			It takes constant execution time to access a fixed word-length lookup table.
			It takes O(log n) execution time and for short division, O($n/\rho + \log \rho$), where n is the word length of the dividend and ρ is the number of processors.
			It needs synchronization for borrowing calculation in parallel.

12	Proposed USP-Awadhoot divider	$Division(Q,R) = f(D_d, D_r)$ $= f(Awadhoot\ matrix, Condition)$ $Awadhoot\ matrix(GQ_n, R_n)$ $= f(GD_d, MD_r, FD)$ $f(GD_d, MD_r, FD)$ $= \{[(R_{n-1} \| GD_{dn})$ $+ (P-Term)_n]$ $- (S-Term)_n\}$ $Condition(Q,R)$ $= f(ND_{dn}, PQ, AQ)$ $Division(Q,R)$ $= (PQ, 0) \quad if\ ND_{dn} = 0$ $Division(Q,R)$ $= [(PQ$ $+ 1), 0] \quad if\ ND_{dn} = D_r$ $Division(Q,R)$ $= (PQ, ND_{dn}) \quad if\ ND_{dn} < D_r$ $Division(Q,R)$ $= [(PQ$ $+ AQ), R_{AQ}] \quad if\ ND_{dn} > D_r$	It is a digit recurrence class, operand data-dependent variable latency divider. It is a non-restoring algorithm based divider but it could be implemented as a restoring algorithm-based divider with particular changes in the Awadhoot matrix and conditions. It implements the novel concept of the dynamic separate scaling operations for the dividend and divisor. The average clock cycle could be reduced with the help of LUTs. No overlapping region in the quotient bit selection logic. Simple conversion logic as the Awadhoot matrix is defined with a simple adder, subtractor, and multiplier.

A pipelined architecture [4, 9, 20, 91, 136] achieves parallelism in sequential architecture with parallel processing. Some or all processes of division algorithms can be pipelined to achieve partial parallel processing. The radix-based SRT division algorithm is among the most often used non-restoring digit recurrence algorithms. The SRT algorithm is widely utilized in serial, parallel, pipelined, or cascading architectures and various applications [15, 20, 50, 70-83, 85-86, 97]. Although the SRT algorithm was the first choice for commercial implementation in the majority of soft and modern processors, like Intel's Pentium processor [92], Xilinx's FPGA controllers [152], and ALU units of complex hardware, it is restricted to specific low radix values, significantly less than 10. Radix-2 and radix-4 are the most implementable formats of the SRT algorithm. The primary reasons for restricting SRT algorithm implementation to specified low radix values are the increase in the criticality of the quotient selection logic and the significant increase in storage area requirements for lookup tables for this logic.

Primarily, low-radix implementation is limited to one or two quotient bits per iteration. In order to decrease division latency, more bits must be retired per cycle. Increasing the radix can improve the cycle time but raises the divisor multiplier formation complexity. The alternative is either a pipelined structure or two-stage lower radix stages merged to generate higher radix dividers through simple staging or, possibly, overlapping the quotient selection logic and partial remainder computation hardware. The use of architecture is not limited or restricted to a particular application. Maximum division algorithm-based dividers can be implemented by a serial, parallel, or pipelined architecture depending on the application's cost, area, and complexity suitability. Generally, improvement in one of these aspects worsens the others; thus, one has to select a particular algorithm based on the specific application requirements. Many researchers have worked on various SRT parameter improvement techniques, such as pre-scaling operands, carry-save remainder, array implementation, truncation, differential look-up

tables, and pre-computed values, but they have not worked on two different performance improvement techniques simultaneously with individual input operands. This gap in the research (i.e., not simultaneously utilizing multiple performance improvement techniques with individual input operands to improve divider implementation) presents an opportunity to develop a new technique or combination of fast or moderate methods that are also area-efficient. I propose a digit recurrence divider based on a state-of-the-art novel USP-Awadhoot algorithm, for improving distinctive divider implementation with moderate operation speeds that are suitable for area critical application. The USP-Awadhoot divider algorithm developed the dynamic separate scaling operations for input operands. In the following sections, I discuss the implementation of a state-of-the-art novel USP-Awadhoot divider, developed according to the ancient theories provided by Vedic mathematics [164] centuries ago. I also discuss the statistical analysis of implementation resources and elaborate on the comparative discussion with different dividers, followed by a conclusion and suggested future work directions.

2.6 Chapter conclusion

A detailed analysis of the different dividers has been presented to understand the needs of the divider circuit block. It helps to decide the fundamentals of the dynamic separate scaling operations. This chapter covers RO1 – Investigate the currently existing divider solutions to understand the different concepts of conversion logic, conceptualize the trade between Area, Speed, and Power, and propose a suitable option or combination of options to develop an efficient divider. Publications I, II, and IV cover a detailed review of the different divider implementations.

3 Design methodology – objective, hypothesis, and algorithm for the proposed divider circuit block implementation

This chapter is based on publications II, III, and IV. All mathematical operations have been implemented by using electronic or digital platforms. However, it is still critical to implement division operations, even though more focus has been put on developing high-performance, faster adders and multipliers than the divider logic. As per Oberman and Flynn's article [15], the electronic implementation of multiplication and addition requires considerably fewer clock/machine cycles and falls into a range of less than ten clock cycles. On the contrary, division operations require more than tens of clock cycles, particularly in the range of 10 to 80 clock cycles. Recursive subtraction or multiplication is at the core of every division algorithm used for the electronic implementation of a divider. Details of the proposed novel USP-Awadhoot divider are discussed further in this chapter.

3.1 Objective

The division operation is the most complex and essential arithmetic operation for digital circuits, computer systems, and embedded systems. Attempts have been made to improve its implementation by optimizing hardware resources or latency cycles. Generally, improvements in one aspect worsen the others, requiring the selection of a particular technique based on application requirements. Implementing division operations in FPGA is necessary because these devices are increasingly employed to develop essential system-on-chip applications or enhance current systems, and indirect division operation results are insufficient. Over the past five years, minimal research has been conducted and presented in the direction of designing a better division algorithm. Most have been developed on the SRT algorithm, based on radix-n and high radix dividers. In the past decade, a few efforts have been based on different theories, alternate physical designs, application-specific parallel computation, and functional iterations, to develop a state-of-the-art algorithm for efficient divider implementation. This has motivated the development of a new technique or combination of improving latency time and implementation area reduction techniques.

3.2 Hypothesis

There are two primary approaches to enhancing the electronic implementation efficiency of a divider circuit block. The first involves optimizing the algorithms that govern the logical data flow and computational processes within the hardware. The second focuses on improving the hardware design, including the interconnectivity of components and the overall architecture of the divider circuit block. The first form of improvement is favored because it is more time and cost-efficient compared to hardware upgrades, which can be up to 100 times more expensive than soft modifications, such as algorithm enhancement. Due to the interdependence of software and hardware changes, better algorithms can be developed based on the best hardware; the best hardware can be developed based on algorithmic requirements, even though we must consider a better trade-off between soft and hardware changes for better improvements. Depending on technological advancements, new algorithms are concurrently being developed with older algorithms to accomplish the same function more effectively. Based on the application requirements,

previous algorithms could be improved, a new algorithm could be constructed, or a new hardware architecture could be created.

Scaling down the operands with a static (fixed) scaling factor is a primary choice in most schemes used to enhance the performance of divider circuit blocks. There may be different methods of calculating the scaling factor but the same factor has scaled down both operands (divisor and dividend). Thus, even after scaling down, the relationship (or ratio) between the divisor and dividend remains the same.

If the dividend is (x) and the divisor is (y) and both the operands are scaled down by a common scaling factor (m) then the relation between the dividend and divisor is expressed as

$$(x \ \rightarrow y) \ = \ (x_m \rightarrow y_m) \tag{1}$$

For example, if the dividend (x) = 500 and divisor (y) = 50, this is scaled down by a common factor (m) = 5. So, the scaled down value of the dividend (x_m) = 100 and divisor $(y_m) = 10$. The ratio of the initial value of the dividend (x) and divisor (y) is given as:

$$\left(\frac{x}{y}\right) = \left(\frac{500}{50}\right) = 10 \tag{2}$$

and the ratio of the scaled-down value of the dividend (x) and divisor (y) is presented as

$$\left(\frac{x_m}{y_m}\right) = \left(\frac{100}{10}\right) = 10 \tag{3}$$

There may be several ways of finalizing the scaling factor but, as the state-of-the-art, the same scaling factor is used to scale down the operands. By scaling the operands, we can reduce their values which, in turn, decreases the number of iterations needed to calculate the quotient bits. However, after a certain point, further scaling becomes impossible because one of the operands reaches its limit, even though the other operand could still be scaled down further. Nevertheless, it increases the area overhead. Thus, in the present research, we hypothesized that a novel concept of the dynamic separate scaling operation or factor could be utilized for operands, lowering the number of iterations necessary for quotient computation and ensuring area reduction.

Additionally, it is believed to be advantageous to divide the initial dividend value into multiple group dividends to ease the quotient bit selection logic. It is also hypothesized that using Vedic sutras can derive a new and constant logic for quotient bit selection. I also considered a hexadecimal system frame structure in developing the quotient selection logic, which is expected to provide easy computation. The following sections describe the novel algorithm for divider implementation, developed based on the above-considered hypothesis. This chapter is concerned with the following research objectives.

- RO2- Develop the theory of conversion logic to implement dynamic separate scaling operations for input operands. Here, a separate scaling operation means using different scaling operations simultaneously for input operands. A partitioning operation is used for the dividend. An operation comprising the "Veshtanam Sutra (by osculation) and Lopanasthabhyam sutra (by elimination and retention)" is used for the divisor. 'Dynamic' refers to the different values of "Flag Digit (*FD*) and Number of Zeros Cancelled (NZC)", used in the Veshtanam Sutra (by osculation) and Lopanasthabhyam sutra (by elimination and retention) depending on the combination of input operands.

- RO3- Divider Algorithm formulation to reduce the criticality of conversion logic by eliminating overlapping regions in quotient selection.

3.3 Introduction to the Vedic sutras

This section describes the basics of some Vedic sutras that were used for developing the novel USP-Awadhoot divider. Vedic Mathematics is an ancient system of mathematics that originated in India, derived from the Vedas, the oldest Indian scriptures. It is a collection of mathematical techniques and shortcuts designed to simplify and speed up calculations. The system was rediscovered in the early 20^{th} century by Swami Bharati Krishna Tirthaji, who compiled and explained these methods in his book "Vedic Mathematics" [164, 167]. Vedic sutras are generally the equations that define relationships between variables or quantities. The following Vedic sutras were used in the development of the novel USP-Awadhoot divider:

- Veshtanam sutra (by osculation)
- Lopana-Sthapanabhyam sutra (by elimination and retention)
- Aanurupyen sutra (proportionately or by suitable ratio)

3.3.1 Veshtanam sutra (by osculation)

The Veshtanam sutra is a key principle in Vedic Mathematics; it emphasizes the concept of 'wrapping' or 'encircling'. The word Veshtanam means "to encircle, enclose, or wrap around" in Sanskrit. This Sutra is often applied in solving equations, particularly in algebra, and in operations involving multiplication, division, and factorization. The Sutra suggests that a solution or simplification can often be achieved by encircling or grouping terms in a convenient way, to simplify calculations. It is especially useful in cases where operations involve recurring patterns or cyclic properties. The sutra can be applied to wrap numbers around a convenient base for easier computation.

3.3.2 Lopana-Sthapanabhyam sutra (by elimination and retention)

The Lopana-Sthapanabhyam sutra is a profound and versatile principle in Vedic Mathematics. The Sanskrit phrase can be broken down as follows:

- Lopana: elimination or removal
- Sthapanabhyam: retention or substitution

Thus, the sutra translates to "by elimination and retention" and it provides a systematic approach to simplifying and solving problems by strategically eliminating and retaining terms or variables.

3.3.3 Aanurupyen sutra (proportionately or by suitable ratio)

The Aanurupyena sutra is an important principle in Vedic mathematics. Its meaning can be derived from the Sanskrit term:

- Aanurupyena: proportionately or by a suitable ratio

This sutra is often applied in mathematical operations where proportions, patterns, or ratios can simplify calculations. It emphasizes solving problems by finding a proportional relationship or a convenient scale or identifying patterns that make the computation easier. It reduces complex division problems into simpler equivalent ratios.

3.4 Introduction to the proposed novel USP-Awadhoot divider circuit block

Digit recurrence division has been proven to be utilized in most processors. Commercial users like IBM, Xilinx, Intel, AMD, Quartus, and HP use many digit recurrence implementations for their processors, as seen in [15, 50, 70, 75, 83, 88, 92-105, 109-118, 152], due to its simple logic for conversion. It gives the root thought of working with digit recurrence division. As per [154], a division is one of the most complex and slowest arithmetic operations performed electronically. Even though division occurs less frequently than other arithmetic operations, an efficient divider is required for optimal system performance. While working with large digital systems, there are new possibilities for reading and writing errors. One common way to solve this problem is to put the binary numbers in a predetermined order [153]. In the proposed novel USP-Awadhoot divider, the Dividend (D_d), Divisor (D_r), Quotient (Q), and Remainder and Residue (R) are presented in hexadecimal format, similar to the concept of using alternative BCD coding in a radix-10 SRT divider implementation [105].

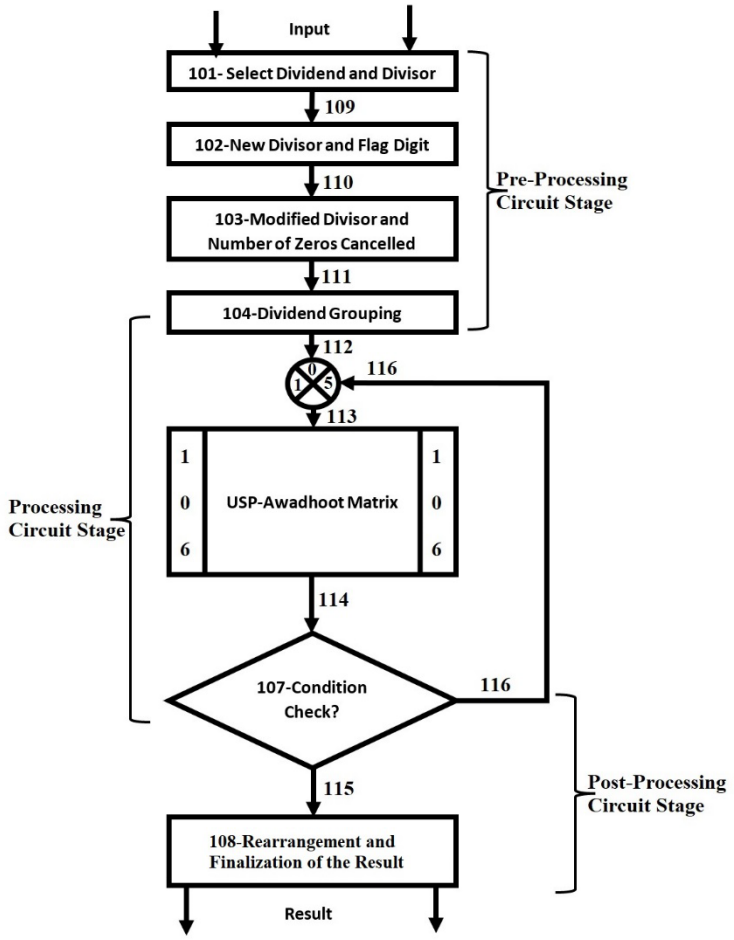


Figure 4. Functional block diagram of the proposed novel USP-Awadhoot divider.

Figure 4 illustrates the functional block diagram of the proposed novel USP-Awadhoot divider circuit block, which consists of three circuit stages: pre-processing, processing, and post-processing. The pre-processing circuit stage consists of number blocks {101 to 104}. Block {101} represents the dividend and divisor selection; block {102} represents new divisor and flag digit calculation; block {103} represents modified divisor calculation and number of zeros cancellation; and block {104} represents a dividend grouping operation. Block {101} accepts the input data (dividend and divisor values) and performs initial processing to verify that the operands are in the proper format for further calculations. In the pre-processing circuit stage, blocks {102 to 104} implement the different scaling operations or factors on the input operands. These different scaling operations, or factors, introduce a nonlinear relationship between the dividend and divisor, effectively reducing the number of iterations required to calculate the quotient. In the proposed novel USP-Awadhoot divider, the dividend grouping circuit plays an essential role in delivering variable latency, as the quotient digit is generated based on dividend grouping, and the size of the dividend groups varies based on the input operand and the number of zeros cancellation (*NZC*) circuits. This variable nature of the dividend group determines the number of input operand bits used to calculate a group quotient. Hence, the output latency is correlated with the distance between input operands. This partitioning dividend and divisor operand flexibility improves the scaling impact and minimizes the instances required to generate a quotient.

The processing circuit stage consists of number blocks {104 to 107}. Block {105} and {107} work as condition checks and control units. Block {106} represents the Awadhoot matrix; a detailed explanation is given in Subsection 3.6. Block {105} manages the data received from the pre-processing circuit stage and condition check unit for the Awadhoot matrix. After performing dividend grouping, iterations, and condition checks, the Awadhoot matrix circuit works on each dividend group in sequence. After performing the last iteration, which is nothing but the last dividend group, all individual dividend group results are provided to the post-processing circuit stage. The post-processing circuit consists of number blocks {107 to 108}. Block {108} represents the rearrangement and finalization of the results. It receives separate quotient bits (hereafter termed group quotient bits) and the last iteration remainder to formulate the final quotient and remainder. Additionally, it generates a controlling signal output that validates the correctness of the division operation performed by the proposed divider circuit block. Detailed explanations of the key terms and working principles of the proposed novel USP-Awadhoot divider are presented in the subsequent sections of this chapter.

3.5 Important terms of the USP-Awadhoot divider circuit block

Pre-processing circuit elements perform input processing and provide data for processing circuit stage elements. This covers input data storage, control, number of zeros cancellation, and modified divisor circuits. The multiple outputs yielded by the pre-processing circuit stage are further fed into the processing circuit stage. The processing circuit stage iteratively constructs the core conversion logic demonstrated by the Awadhoot matrix. This Awadhoot matrix consists of dividend groups, P-term, S-term, net dividend, group quotients, and the remainder, arranged within each iterative circuit stage. At the end of the processing circuit stage, all individual dividend group quotients and the remainder are passed to the post-processing stage. In the post-processing stage, all the individual dividend group quotients are re-arranged to form the final quotient and remainder. A detailed description of the three stages is

provided but it is essential to understand the vital terms or elements used in these three circuit stages of the USP-Awadhoot divider circuit implementation. The important signals and terms used in the proposed novel USP-Awadhoot divider are:

$$Dividend\ (D_d) = Divisor\ (D_r) \times Quotient\ (Q) + Remainder\ (R) \qquad (4)$$

- $D_d = [d_1\ d_2 \dots\dots\dots\ d_k]$; where "$D_d$" represents dividends with a maximum size of "*k*" digits.
- $Q = [q_1\ q_2 \dots\dots\dots\ q_k]$; where "*Q*" represents the quotient with a maximum size of "k" digit.
- $D_r = [d_1\ d_2 \dots\dots\dots\ d_k]$; where "$D_r$" represents the divisor with a maximum size of "*k*" digits.
- $R = [r_1\ r_2 \dots\dots\dots\ r_k]$; where "*R*" represents the remainder with a maximum size of "k" digits.
- $ND_r = [ndr_1\ ndr_2 \dots\dots\dots\dots ndr_m]$; where "$ND_r$" represents the "New Divisor" with a maximum size of "*m*" digits. The range is defined as: "$k-1 \leq m \leq k+1$".
- $FD = [fd_1\ ]$; where "*FD*" represents the "Flag Digit" with the maximum size of a single digit in a fixed range of [1,2……9].
- $MD_r = [md_1 md_2 \dots\dots\dots\ md_p]$; where "$MD_r$" represents the "Modified Divisor" with a maximum size of "*p*" digits. The range is defined as: "$p \leq k-1$".
- $NZC = [nzc_1\ nzc_2 \dots\dots\dots\ nzc_p]$; where "*NZC*" represents the "Number of Zeros Cancelled" with a maximum size of "*p*" digits. The range is defined as "$p \leq k-1$".
- $ND_d = [ndd_1\ ndd_2 \dots\dots\dots\ ndd_k]$; where "$ND_d$" represents the "Net Dividend" with a maximum size of "*k*" digits.
- $G_rD_d = [gdd_1\ gdd_2 \dots\dots\dots\ gdd_k]$; where "$G_rD_d$" represents the "Gross Dividend" with a maximum size of "*k*" digits.
- $GD_d = [gd_1\ gd_2 \dots\dots\dots\ gd_k]$; where "$GD_d$" represents the "Group Dividend or Dividend Group" with a maximum size of "*k*" digits.
- $GQ_n = [gq_1\ gq_2 \dots\dots\dots\ gq_k]$; where "$GQ_n$" represents the "Group Quotient" with a maximum size of "*k*" digits.
- $(P-Term) = [(p-term)_1\ \dots\ (p-term)_k]$; where "$(P-Term)$" represents the product term with a maximum size of "*k*" digits.
- $(S-Term) = [(s-Term)_1\ \dots\ (s-Term)_k]$; where "$(S-Term)$" represents the sum term with a maximum size of "*k*" digits.
- *PQ* $= [pq_1\ pq_2 \dots\dots\dots\ pq_k]$; where "*PQ*" represents the partial quotient with a maximum size of "*k*" digits.
- *AQ* $= [aq_1\ aq_2 \dots\dots\dots\ aq_k]$; where "*PQ*" represents the additional quotient with a maximum size of "*k*" digits.

- $R_{AQ} = [r_{aq1}\, r_{aq2} \ldots\ldots\ldots r_{aqk-1}]$; where "$R_{AQ}$" represents the remainder generated during the calculation of the additional quotient with a maximum size of "k" digits.

3.6 The Awadhoot matrix for iterative circuit elements of the processing circuit stage

Figure 5 illustrates the particular arrangement of the elements of the processing circuit stage of the proposed novel USP-Awadhoot divider. The structure is defined as the Awadhoot matrix. Awadhoot means a 'different than normal' or 'unique' arrangement. The Awadhoot matrix provides a computational arrangement of various aspects of the processing circuit stage of the proposed novel USP-Awadhoot divider. The Awadhoot matrix is a vital element of the proposed novel USP-Awadhoot divider. Figure 5 shows that each column represents an individual iterative circuit stage, and each row represents the elements of the corresponding iterative circuit stage. The logical interconnection between different aspects of the Awadhoot matrix is described in the inset picture of Figure 5. Depending on the hardware architecture to be used, we can either use a single set or multiple sets of iterative circuit elements. The Awadhoot matrix arrangement is composed of the previous remainder (R_{n-1}), group dividend (GD_{dn}), previous iteration group quotient (GQ_{n-1}), gross dividend (G_rD_{dn}), flag digit (FD), modified divisor (MD_r), net dividend (ND_{dn}), the present quotient (GQ_n) and the present remainder (R_n).

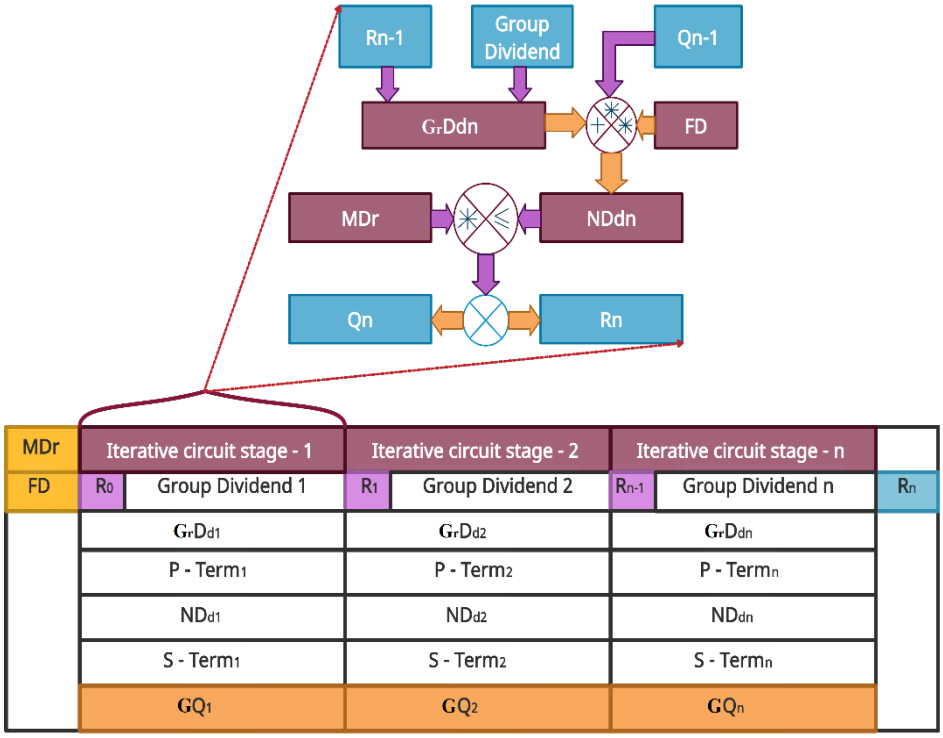


Figure 5. The Awadhoot matrix.

The hardware requirements of the proposed divider circuit depend on the possible number of dividend groups created during the pre-processing circuit stage of the proposed divider; the maximum number of dividend groups is related to the maximum width of available operands. This Awadhoot matrix arrangement provides a detailed structure of the processing circuit stage, which can be realized by serial, parallel, or pipeline hardware architecture. During the execution of the first iterative circuit stage, both the previous remainder (R_{n-1}) and the previous iteration group quotient (GQ_{n-1}) are assumed to be zero, to avoid computation errors. Upon completing the first iteration, the generated remainder and group quotient are passed to the next iteration circuit stage, where they serve as the previous remainder (R_{n-1}) and the previous iteration group quotient (GQ_{n-1}). This process is repeated for each subsequent iteration, until the final iteration circuit stage is reached.

A detailed description of the Awadhoot matrix is given in the patent application [166]. In short, the gross dividend (G_rD_{dn}) is derived from the previous remainder (R_{n-1}) and the present value of the group dividend at the first level of the iterative circuit stage of the Awadhoot matrix. Further simple addition and multiplication operations are performed with gross dividend (G_rD_{dn}), previous iteration group quotient (GQ_{n-1}) and flag digit (FD), to derive the value of the net dividend (ND_{dn}), which is indicated by the *P*-term terminology in the Awadhoot matrix. Further multiplication operations are performed, depending on the condition of the present net dividend (ND_{dn}) value in comparison with the value of the modified divisor (MD_r), to obtain the value of the *S*-term. Depending on the comparison, the final value of the group quotient (GQ_n) and the present remainder (R_n) are calculated and presented for the next iterative circuit stage or post-processing circuit stage. During the execution of the post-processing circuit stage, all individual group quotient values are re-arranged together, along with the associated weights, to form the final quotient value. Later, this final quotient and the remainder values are displayed or transmitted to other circuits if necessary.

In the USP-Awadhoot divider, the dividend grouping circuit is crucial for providing variable latency features. Unlike others, the proposed USP-Awadhoot divider converts the dividend into group dividends, which are not required to add up to the primary dividend value. Dividend = (group dividend 1, group dividend 2, …, group dividend n), where Dividend ≠ (group dividend 1+group dividend 2+⋯+group dividend n). For example, if the dividend is 1055, it can be partitioned into two group dividends, such as 10 and 55, where the sum of these two group dividends, i.e., 10 + 55 = 65, is not equal to the original dividend value. This dividend grouping mechanism works as a separate dividend scaling factor or operation. After completing the dividend grouping, iterate through each group sequentially, performing calculations and condition checks on each. After the final iteration, which is the final set of dividends, the post-processing circuit stage calculates the final quotient and remainder. I considered the hexadecimal number system to implement the proposed system due to its ease of use in digital systems and computer applications. In digital electronics, hexadecimal numbers give better readability and provide a fixed bit size frame structure to represent each decimal number in digital form.

The binary representation of decimal numbers or digits provides multiple modes of representation. Sometimes, it can be represented by a one-bit equivalent binary number or multiple-bit binary number; whereas, in the case of hexadecimal representation, every hexadecimal digit is defined as a frame of four binary bits. The fixed frame of representing hexadecimal numbers in digital or binary form provides better support to perform operations like shifting, comparing, and giving a simple logic for quotient bit

selection in the processing circuit stage. A hexadecimal system also simplifies internal operations, such as concatenating digits in digital computation. A binary system could also be used but hexadecimal numbers give the advantage of working with four bits per digit each time. In the binary system, the minimum number of bits to be considered for computation is one; in the hexadecimal system, four binary bits are used, which provides greater clarity for comprehending the computation process performed on a digital system containing long bitstream data.

As expressed in equations (5) to (7), the conversion of any digit value of any number system into a single digit by the repetitive addition of all digits is called 'Beejank' or 'Digital root'. Beejank does not indicate a deficiency in minimum (zero) and maximum numbers. If the place of a digit(s) in a number is/are interchanged, then this change is not indicated by the Beejank.

$$(F89A0BCD)_{16} = (F + 8 + 9 + A + 0 + B + C + D)_{16} \quad (5)$$

$$= (4E)_{16} = (4 + E)_{16} = (12)_{16} = (3)_{16} \quad (6)$$

Beejank calculations are helpful for confirming the correctness of the quotient calculated in the Awadhoot matrix. As shown in equation (7), if the left side value equals the right side value, then it is confirmed that the calculated quotient is correct.

$$Beejank(D_d) = Beejank(Q) * Beejank(D_r) + Beejank(R) \quad (7)$$

For example, let us consider a dividend D_d=9216, a divisor D_r=72, a calculated quotient (Q)=128, and a remainder (R)=0. The Beejank values are as follows: $Beejank(9216)$=9, $Beejank(72)$=9, $Beejank(128)$=2 and $Beejank(0)$=0.

$$Beejank(9216) = Beejank(128) * Beejank(72) + Beejank(0) \quad (8)$$

$$9 = 2 * 9 + 0 = 18 = 9 \quad (9)$$

Table 3. Hexadecimal representation of addition.

Add	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	A
2	2	3	4	5	6	7	8	9	A	B
3	3	4	5	6	7	8	9	A	B	C
4	4	5	6	7	8	9	A	B	C	D
5	5	6	7	8	9	A	B	C	D	E
6	6	7	8	9	A	B	C	D	E	F
7	7	8	9	A	B	C	D	E	F	10
8	8	9	A	B	C	D	E	F	10	11
9	9	A	B	C	D	E	F	10	11	12

The operation of Beejank (digital root) and alternate representation of addition and subtraction, which we perform during the iteration of the processing circuit stage, exhibits great ease in the quotient bit selection logic developed with the hexadecimal system. Table 3 expresses the hexadecimal representation of addition. Hence, we consider a hexadecimal number system in quotient bit selection logic, to confirm the correct selection of the quotient bit in a particular iteration.

3.7 The working principle of the proposed novel USP-Awadhoot divider circuit block

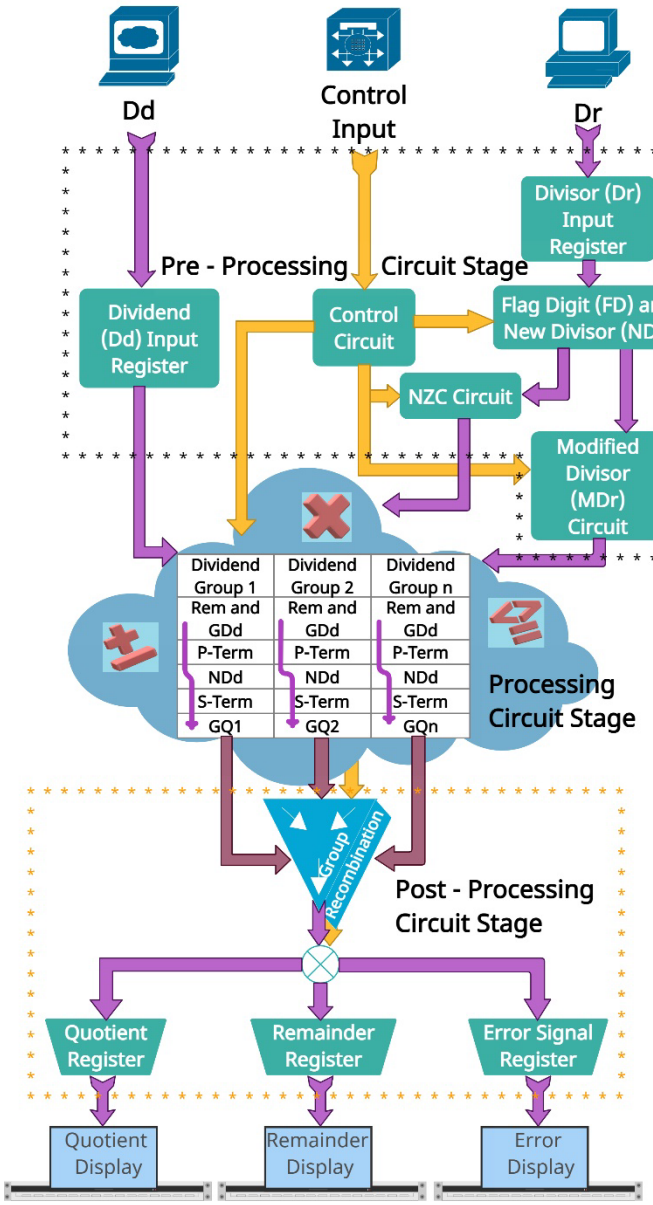


Figure 6. Schematic block diagram of the proposed USP-Awadhoot divider.

$$Division(Q,R) = f(D_d, D_r) = f(Awadhoot\ matrix, Condition) \quad (10)$$

$$Awadhoot\ matrix(GQ_n, R_n) = f(GD_d, MD_r, FD) \quad (11)$$

$$f(GD_d, MD_r, FD) = \{[(R_{n-1}||GD_{dn}) + (P-Term)_n] - (S-Term)_n\} \quad (12)$$

$$Condition(Q,R) = f(ND_{dn}, PQ, AQ) \quad (13)$$

where *PQ* is the partial quotient, *AQ* is the additional quotient and R_{AQ} is the remainder generated during the calculation of the additional quotient. Therefore, after the last iteration of the Awadhoot matrix, based on the condition function, the final quotient and remainder value are calculated and represented as

$$Division(Q,R) = (PQ, 0) \qquad if\ ND_{dn} = 0 \quad (14)$$

$$Division(Q,R) = [(PQ+1), 0] \qquad if\ ND_{dn} = D_r \quad (15)$$

$$Division(Q,R) = (PQ, ND_{dn}) \qquad if\ ND_{dn} < D_r \quad (16)$$

$$Division(Q,R) = [(PQ+AQ), R_{AQ}] \qquad if\ ND_{dn} > D_r \quad (17)$$

Equations (10) to (17) represent the proposed USP-Awadhoot divider algorithm formulation. This indicates that the actual division is a function of the input operands, depending on the conditions explained in the function of the Awadhoot matrix and controlling conditions. Figure 6 illustrates the schematic block diagram of the proposed novel USP-Awadhoot divider derived from the functional block diagram presented in Figure 4. The key functional block {101} represents the dividend and divisor input register with the control circuit block, while functional block {102} handles the flag digit (*FD*) and the new divisor (ND_r) calculation circuit. Functional block {103} manages the modified divisor (MD_r) and the number of zeros cancelled (*NZC*) calculation circuit, while functional block {104} performs the dividend grouping (GD_d) operations. Functional blocks {105, 106, and 107} execute the Awadhoot matrix calculations and condition checks, and functional block {108} handles the quotient and remainder re-arrangement, along with the output display circuit.

After providing all inputs, at numbered block {101}, the pre-processing circuit stage obtains the divisor (D_r) and dividend (D_d) with a maximum word size of "*k*" digits. The width of the dividend and divisor determines the circuit hardware requirements. Prior to storing inputs in the dividend and divisor operand register, the divisor (D_r) and dividend (D_d) operands undergo an input normalizing process to verify that the width size of the input operands is within permissible limits and in the required frame format. The use of hexadecimal numbers simplifies the implementation of this phase. This stipulation is not a limitation. Multiple number systems were utilized in SRT divider implementations, to reduce the criticality of the input circuitry. One of the best ways to show this is with BCD numbers, as shown in [105], which explains how BCD numbers are used to implement the radix-10 SRT divider. A hexadecimal number format is used with the proposed algorithm, to offer a robust framework for electronic implementation. It is not limited to hexadecimal number systems and may also be used with binary, decimal, and octal number systems. The controlling signal circuit generates reference signals for regulating the individual elements of the three circuit stages of the proposed divider, then the divisor (D_r) undergoes the condition check for invalid conditions, i.e., dividing by zero. This would indicate an error signal at numbered block {108}, as indicated by signal 115, and this would be redistributed for display or transmission in the post-processing circuit stage, upon detecting the invalid condition. In a false scenario,

signal 109 passes the divisor (D_r) to numbered block {102}, where the circuit acquires the flag digit (*FD*) and new divisor (ND_r) and follows the basic concept of obtaining the flag digit (*FD*) and new divisor (ND_r).

Later, at numbered block {103}, the flag digit (*FD*) and new divisor (ND_r) are used to obtain the modified divisor (MD_r) and the number of zeros cancelled (*NZC*). This step follows the basic concept of obtaining the MD_r and the *NZC*. The MD_r works as a separate divisor scaling factor or operation. Furthermore, the *FD* and *NZC* values are delivered to the numbered block {104} by the 111 path. At this stage, dividend sectioning/regrouping is carried out, and dividend groups are distributed based on the *NZC* value generated in the previous stage. Unlike the various SRT implementations that utilize operand pre-scaling or truncation [92, 134, 147], a fixed number of dividend sectioning or partitioning operations [4], the proposed divider performs a cross-combination of pre-scaling of the divisor and sectioning or partitioning of the dividend. This simultaneous cross-implementation of two different procedures yields the innovative concept of the dynamic separate scaling operation or factor for the dividend and the divisor.

As discussed, the hardware requirements depend on the operand size; the maximum number of elements in an iterative circuit stage never exceeds the maximum operand size. It is recommended that, if the operand size is 8 bits, a maximum of eight iterative circuit stages may be used and yet, the number of iterative circuit stages used in a particular conversion depends on the *NZC* value. Similar to the variable latency class algorithms, the dynamic nature of iterative circuit stages provides flexible conversion clock cycles for every dividend-divisor combination, with the possibility of a variable quotient bit retiring rate in different iterations or some iterations requiring less execution time. This results in different conversion times in different sets of dividends and divisors.

Once the *NZC* value has been determined, the circuit completes the pre-processing stage and sends the data to numbered blocks {105-107}, to arrange the dividend, MD_r, $a$and *FD* in separate dividend groups, as shown by the Awadhoot matrix. As illustrated in Figure 6, the Awadhoot matrix is employed in the processing circuit stage of the proposed divider, to calculate the group quotient and remainder. The number block {106} represents the iterative circuit steps. As previously mentioned, the maximum number of iterative circuit steps cannot exceed the width of the operand. The condition checker, numbered block {107}, confirms the end of computation in the iterative circuit step of the processing circuit stage. Once block {107} completes the calculation and the data is passed to the post-processing circuit stage at block 108. Figure 6 shows the group re-arrangement circuit, followed by a distribution circuit, allowing separate visualization or transmission of the quotient (*Q*) and the remainder (*R*). After computing the Awadhoot matrix block {106}, via equation (11), the individual group quotients are re-arranged as per their relative weights, to generate the final group quotient (*Q*). The final residue or remainder is obtained from the last iterative circuit step, depending upon the conversion status.

- First: Net Dividend (ND_d) = 0. Shows that the dividend (D_d) is completely divisible by the divisor (D_r), with the Remainder (*R*) = 0 and the Quotient (*Q*) = the Partial Quotient (PQ_n) formed by concatenating the individual group quotient (GQ_n).

- Second: Net Dividend (ND_d) = Divisor (D_r). Shows that the dividend (D_d) is completely divisible by the divisor (D_r) with the Remainder (*R*) = 0 and the Quotient (*Q*) = the Partial Quotient (PQ_n) + 1.
- Third: Net Dividend (ND_d) > Divisor (D_r). The Remainder (*R*) = R_{AQ} is the value obtained during the calculation of the additional quotient (AQ) and the Quotient (*Q*) = the Partial Quotient (PQ_n) + the Additional Quotient (AQ), where the Additional Quotient (AQ) is derived by initializing the count to zero and subtracting the Divisor (D_r) from the last iteration Net Dividend (ND_d) number, incrementing the count by one. Continue the same process until we get a subtraction result of zero or less than the divisor (D_r).
- Fourth: Net Dividend (ND_d) < Divisor (D_r). Remainder = value of the last iteration ND_r and the Quotient (*Q*) = the Partial Quotient (PQ_n).

In the postprocessing circuit stage, the final step of the proposed divider rearranges the individual group quotient (GQn). At the end of the division process, the final quotient and remainder are generated, accompanied by an error signal indicator to verify the accuracy of the data. Figure 7 presents a detailed explanation of the data flow and critical circuit path of the proposed divider circuit speculated in Figure 6. Inputs include the dividend (D_d) and divisor (D_r), in addition to the control inputs (*reset, fd_enable, and clock*). The logic flow state diagram of the proposed divider circuit contains twenty-four states, of which twenty-three are functional logic states, and one is an error state. The green track represents the critical path of the proposed divider circuit implementation, the red track represents the feedback path, and the purple track represents the conditions. Multiple iterations must be performed depending on the conditions indicated by the purple track. Details of individual states are given as:

- ***St0:*** This is the initial state of the proposed divider circuit indicating the idle condition of operation, where the outputs are deactivated, and the previous rest and quotient values are set to zero prior to initializing circuit operation. Both *fd_enable* and *RST* control signals are active high signals. The divider circuit maintains its idle state St0 until the *fd_enable* control signal is turned to logic one, and *RST* is turned to logic 0, indicating that when the circuit overcomes its idle state and continues the process. If the *RST* control input is activated (*RST = 1*) during operations, the course returns to the idle St0 state until the *RST* control input is deactivated and a new *fd_enable* is applied to the circuit. No input data is transmitted until *fd_enable* is applied to the circuit; information is available on the input data lines.
- ***St1:*** This logic flow state represents dual responsibilities to delineate. After the activation of the *fd_enable* control signal, the circuit allows the data available at the input data lines to be stored at input registers. The most significant bits (MSB) and the less significant bits (LSB) are stored as an array of hexadecimal integer elements for the dividend and divisor. *FD* is formulated by only working on the hexadecimal LSB part of the divisor. This process further extends and computes the value of ND_r. Another responsibility is to investigate an invalid condition, which is considered to be divided by zero. If the invalid condition appears in the existing computation, the circuit activates the error signal state in the logic flow represented as the *StEr* signal. In the absence of the invalid division condition, the computation continues to execute the next stage (*St3*).

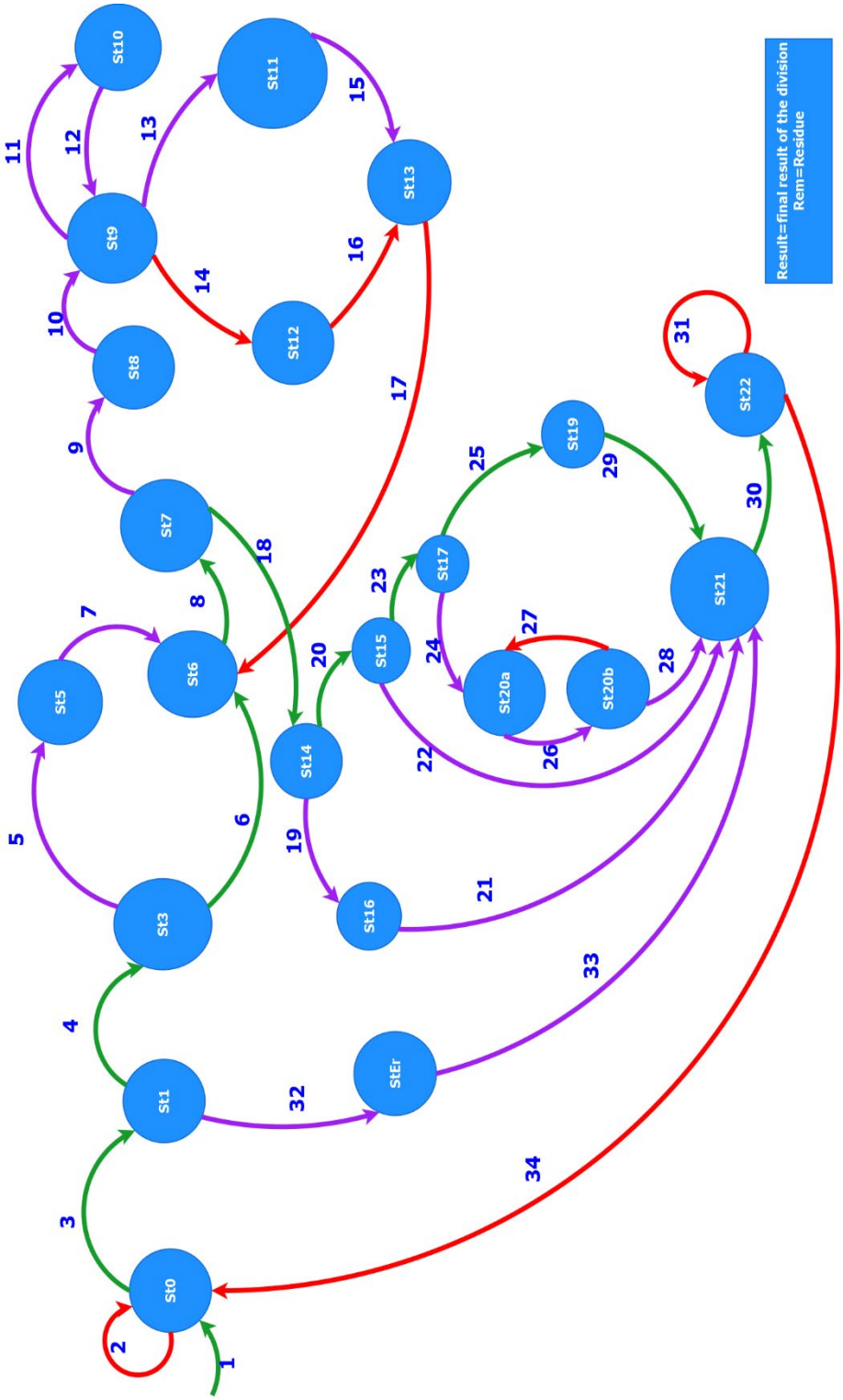


Figure 7. The logic flow state diagram of the proposed USP-Awadhoot divider.

- ***St2:*** This logic flow state is not used for computation; instead, it is used as a buffer state and reserved for future computation improvements with extended operand widths.
- ***St3:*** This logic flow state represents the dual responsibilities of obtaining values of *NZC* and MD_r from the provided operands. Depending on the value of *NZC,* there are one, two, or multiple dividend groups (GD_{dn}). The formation of dividend groups (GD_{dn}) is based on the relative weights of the hexadecimal operand. Every operand group is represented as an iterative circuit stage. Thus, the hardware requirement depends on the number of dividend groups expected to be computed based on the provided operand width. If the MD_r value is greater than the first iterative circuit stage (GD_{dn}) value, then the next logic flow state to be executed is *St5*, otherwise *St6* is selected.
- ***St4:*** This logic flow state is not used for computation; instead, it is used as a buffer state and reserved for future dynamic computation improvements with extended operand widths.
- ***St5:*** Every iterative circuit stage is associated with an individual group quotient (GQ_n) and partial group quotient. The partial group quotient precedes the group quotient in order to store partial results during individual iterative circuit stage computation, with the idle value set to zero. The residue is the first element of (GD_{dn}). Upon completion, it performs the next state, *St6*.
- ***St6:*** This reflects the concatenation effect in computing the (GD_{dn}) value. Upon completion of the computation of (GD_{dn}) the next logic flow state is *St7*.
- ***St7:*** This iterative circuit stage computes the value of ND_d. *G_Q* is an array where an individual group quotient(GQ_n) is stored. Upon completion, if the counter is greater than the number of groups, it performs the next state, *St14;* otherwise, the next state is *St8*.
- ***St8:*** This iterative circuit stage maintains the counter value and acts as a buffer stage. Upon completion, it performs *St9* as the next logic flow state.
- ***St9:*** This iterative circuit stage computes the *P–term* value. If the value of *P–term* < the value of ND_d then the next state is *St10,* and if the value of *P–term* > the value of ND_d then the next state is *St11,* else *St12*.
- ***St10:*** This iterative circuit stage maintains the counter value and acts as a buffer stage. Upon completion, it performs *St9* as the next logic flow state.
- ***St11:*** This iterative circuit stage computes the iterative circuit stage's expected residue. Upon completion, it performs *St13* as the next logic flow state.
- ***St12:*** This iterative circuit stage is performed if the residue is zero. Upon completion, it performs *St13* as the next logic flow state.
- ***St13:*** This iterative circuit stage maintains the counter value and acts as a buffer stage. Upon completion, it performs *St6* as the next logic flow state.
- ***St14:*** This iterative circuit stage executes an additional quotient's computation whose idle value is set to 0. Upon completion, if ND_d is zero, it performs *St16;* else, it performs *St15* as the next state.

- ***St15:*** This state computes the value of residue and additional quotient. If $ND_d = D_r$ then it performs *St21;* else, it performs *St17* as the next state.
- ***St16:*** This state computes residue value and additional quotient in the standard case, then it performs *St21* as the next state.
- ***St17:*** This state decides the comparative study of the ND_d Value. If it is less than D_r then it performs *St19* as the next state; else, it performs *St20a* as the next state.
- ***St18:*** This state acts as a buffer stage and is not used for active computation. Upon completion, it performs *St9* as the next logic flow state.
- ***St19:*** This state computes the value of the additional quotient when the residue value equals the ND_d value. Upon completion, it performs *St21* as the next logic flow state.
- ***St20a:*** This state acts as a buffer stage and is not used for active computation. Upon completion, it performs *St20b* as the next logic flow state.
- ***St20b:*** This state computes residue value and performs St21 as the next state upon completion.
- ***St21:*** This state computes the final values for quotient and residue as per the display, transfer, or storage requirements.
- ***St22:*** This state acts as a buffer stage and is not used for active computation. It is used to transmit results to the output data lines.
- ***StEr:*** This logic flow state acts as an error indicator, especially indicating invalid computation. It is an active high logic controlling signal endorsing the correct calculation at the end.

3.8 Summary of the proposed USP-Awadhoot divider realization

The proposed divider consists of three circuit parts: the pre-processing circuit stage, the processing circuit stage, and the post-processing circuit stage. All the data operands and control inputs are connected to the pre-processing circuit stage of the proposed divider. Upon receiving input operands and control signals, the pre-processing circuit stage performs the preliminary action of generating data for the processing circuit stage. During the pre-processing circuit stage application or operation, the Modified divisor (MD_r) and dividend grouping, based on *NZC,* works as a separate scaling factor or operation for the divisor and dividend. It helps to reduce the distance between the dividend and divisor beyond the linear relation. The proposed divider converts the dividend into group dividends, unlike others that are not required to add to the primary dividend value. Dividend = (group dividend 1, group dividend 2, …, group dividend n) where dividend ≠ (group dividend 1+group dividend 2+⋯+group dividend n). The execution of the processing circuit stage depends on the Awadhoot matrix, which derives the relation between variably scaled operands and provides quotient bit selection logic. The last step of the proposed divider re-arranges the individual group quotient (GQ_n) in the post-processing circuit stage. Upon completing a division, the final quotient and remainder are available, along with an error signal, to indicate the correctness of the division and presented data.

In Summary, the proposed novel USP-Awadhoot divider circuit implementation generally has 11 steps, depending on the input operands' values, which are as follows:
Step 1 – Define Dividend (D_d) and divisor (D_r).
Step 2 – Derive New Divisor (ND_r) and Flag Digit (*FD*).
Step 3 – Obtain Modified Divisor (MD_r) and Number of Zeros (*NZC*).
Step 4 – Dividend Grouping.
Step 5 – Arrange the Awadhoot Matrix. At the beginning of the 1st iteration, the remainder and previous quotient values are equal to 0, considering the idle condition at the start.
Step 6 – Start Iteration 1 circuit by checking that MD_r> value from 1st group dividend (GD_d) and derive gross dividend ($G_r D_{dn}$) by concatenating the remainder.
Step 7 – Derive p-term from the previous iteration quotient; if it is the 1st iteration, then the value of the last quotient is considered to be an idle condition.
Step 8 – Derive net dividend (ND_d), check for a positive value, and compare MD_r value with ND_d value to get group quotient (GQ_n) value and group remainder value. The group remainder generated in the current iteration acts as a carry-forward value for the next iteration.
Step 9 – Every iteration circuit stage contains steps 6 to step 8.
Step 10 – In the last iteration circuit stage, check the value of ND_d and validate the conversion.
Step 11 – Re-arrange the group quotient values and residual values from the iteration circuit stages to provide valid quotient and remainder values.

3.9 Chapter conclusion

A detailed explanation of the basic working concept of the proposed novel USP-Awadhoot divider is described in this chapter. Modified divisor (MD_r), flag digit (*FD*), and the group dividend (GD_{dn}) operations of the pre-processing circuit stage implement the dynamic separate scaling operations for input operands. In the processing circuit stage, the Awadhoot matrix calculates the group quotients (GQ_n), while the post-processing circuit stage provides a clear process of selecting the final quotient based on the group quotient (GQ_n), partial quotient (PQ), and additional quotient (AQ) values without critical overlapping regions.

This chapter covers RO2 – "Develop the theory of conversion logic to implement dynamic separate scaling operations for input operands". Here, a separate scaling operation means using one conversion operation for the dividend and another conversion operation (a different one in the general case) for the divisor. 'Dynamic' refers to the resulting different scaling operations, depending on the input operand value combinations. RO3 – "Divider algorithm formulation to reduce the criticality of conversion logic by eliminating overlapping regions in quotient selection". Publications II, III, and IV cover detailed information on the proposed novel USP-Awadhoot divider.

4 Complex division by Baudhayan-Pythagoras triplet method using a novel USP-Awadhoot divider

This chapter is based on publication IV. Complex number arithmetic computation is crucial in electrical and electronic applications, such as signal processing, control theory, microwave systems, complex orthogonal transformations, astronomy, automatic gain control (AGC) systems, and demodulators in receivers [46-49, 56, 155-157]. A complex number is represented as a combination of real and imaginary parts. Its real and imaginary parts must be treated separately, making it very complicated to perform arithmetic operations on a complex number. It makes a complex divider critical and space-intensive, which could limit its hardware implementation. There have been several attempts to implement different logical approaches, by recommending an alternative number system to represent a complex number as a unique and combined entity instead of base 2. Examples of this are, the quarter-imaginary number system with a $2j$, -4, $(-1+j)$ and $j\sqrt{2}$ base, a complex binary number system with a $(-1+j)$ base, and a redundant complex number system [158-159]. The main problem associated with the quarter-imaginary number system with $2j$, -4, and $(-1+j)$ base is to derive a definitive division process where the quarter-imaginary number system with $j\sqrt{2}$ base partially generates the solution. This is because the even power of the base generates the real part and the odd power of the base generates the imaginary part. A complex binary number system with a $(-1+j)$ base and a redundant complex number system requires more complex conversion logic for division, resulting in higher area requirements for implementation [158-162].

As the current state-of-the-art, we must implement two sets of dividers for the real and imaginary components of complex numbers in the case of a complex divider. A software or hardware divider forms complex numbers based on the conventional formula mentioned in equation (19), where z_1 and z_2 are two complex numbers that may lead to overflow or underflow conditions when the operands are near the extreme ends of the representable range [46].

$$z_1 = x_1 + iy_1 \; and \; z_2 = x_2 + iy_2 \tag{18}$$

$$\frac{z_1}{z_2} = \frac{(x_1x_2 + y_1y_2)}{(x_2^2 + y_2^2)} + \frac{(x_2y_1 - x_1y_2)i}{(x_2^2 + y_2^2)} \tag{19}$$

There have been several attempts to implement dividers such as digit recurrence SRT dividers and functional iteration-based multiplicative dividers, which resemble the Newton-Raphson and Taylor series. These dividers require two separate dividers to perform the division on the real and imaginary parts of the complex number, giving rise to critical quotient selection logic and extra overhead, to generate the final quotient and remainder in complex numbers. SRT base radix-2 divider implementation is discussed in [46] but it is restricted to low radix values, due to the impracticable quotient digit selection logic. In the case of functional iteration dividers, the correctness of the result depends on the closeness of the reciprocal value selected in the initial iteration. A pre-scaled divider [46, 163], where the divisor and dividend are multiplied by the same scaling factor so that the resultant divisor must be in close proximity to unity, is one of the best approaches for a high-radix complex divider. This method has the primary drawback of requiring an additional full-width divider for calculating the scaling factor. In this chapter, I discuss the method of complex division based on the

Baudhayan-Pythagorean triplet method and the proposed novel USP-Awadhoot divider circuit block. The use of the Baudhayan-Pythagoras triplet algorithm is possible because of the geometric properties of the complex numbers, which can be used to represent them via real and imaginary axis. The proposed complex division implementation is partitioned into three parts.

The Baudhayan-Pythagorean triplet algorithm is used for the input circuit stage, ensuring the separation of the real and imaginary parts of complex numbers for further calculation. The second stage consists of a novel USP-Awadhoot divider circuit block, which divides the real and imaginary parts of the complex number. The third stage involves rearrangement, representing the final results in complex numbers. The Pythagorean theorem was known long before Pythagoras (570–500/490 BCE); Baudhayan (800–740 BCE) is said to be the pioneer of the Pythagorean theorem. Baudhayan formulated the relation between the hypotenuse and other sides of a triangle, in terms of the area, in his book titled "Baudhāyan Śulbasûtra" [164]. In contrast, Pythagoras presented proof of the relationship between the hypotenuse and other sides of a triangle in terms of length [164-165], giving the equation:

$$Bodhayan - Pythagorean\ Triplet\ (x, y, z) = T\ (x, y, z) \tag{20}$$

Here $T\ (x, y, z)$ is represented as the area of the square formed by hypotenuse or larger side and (z) equals to the area of squares formed by the first side (x) plus the area of squares formed by the second side (y) of a triangle. The Vedic formula proportionately tells us that if one triplet is a multiple or sub-multiple of another triplet, then they are called equal triplets because the triangles of these triplets have the same shape and angles, e.g., $5, 12, 13 = 10, 24, 26 = 25, 60, 65$. When transposing the first two triplet elements, they get converted into a complementary triplet, e.g., 3,4,5 & 4,3,5 are complementary triplets after transposing 3 and 4. Similarly, it is possible to apply addition, subtraction, multiplication, and division operations to triplets, to solve more complex computational problems.

4.1 Complex division by Baudhayan-Pythagorean triplet method using the proposed USP-Awadhoot divider

A complex number is a number of the form $(x + iy)$, where *x* and *y* are any real numbers, and *i* is called an imaginary unit, where $i = \sqrt{-1}$ or $i^2 = -1$. x is the real part coefficient of a complex number and y is the imaginary part coefficient.

As $i = \sqrt{-1}$ or $i^2 = -1$, and based on the proposed novel approach, we can correlate the Baudhayan-Pythagorean triplet $T\ (x, y, z)$ function with the complex number, representing a given complex number in terms of $T\ (x, y, z)$. The real and imaginary coefficients of a given complex number are represented by the first two variables of a triplet, e.g., the complex number $r = x + iy$ can be represented in the Baudhayan-Pythagorean triplet $T\ (x, y, z)$. The following equations are used to develop the input circuit stage.

$$r_1 = x_1 + iy_1\ and\ r_2 = x_2 + iy_2 \tag{21}$$

$$T(r_1) = f\ (x_1, y_1, z_1) \tag{22}$$

$$T(r_2) = f\ (x_2, y_2, z_2) \tag{23}$$

$$z_2^2 = (x_2^2 + y_2^2) \tag{24}$$

As per the equations (19) and (22) to (24), the triplet of the division can be found as:

$$\left[T\left(\frac{(r_1)}{(r_2)}\right)\right] = \frac{f\ (x_1, y_1, z_1)}{f\ (x_2, y_2, z_2)} = [(x_1x_2 + y_1y_2), (x_2y_1 - x_1y_2), z_2^2] \tag{25}$$

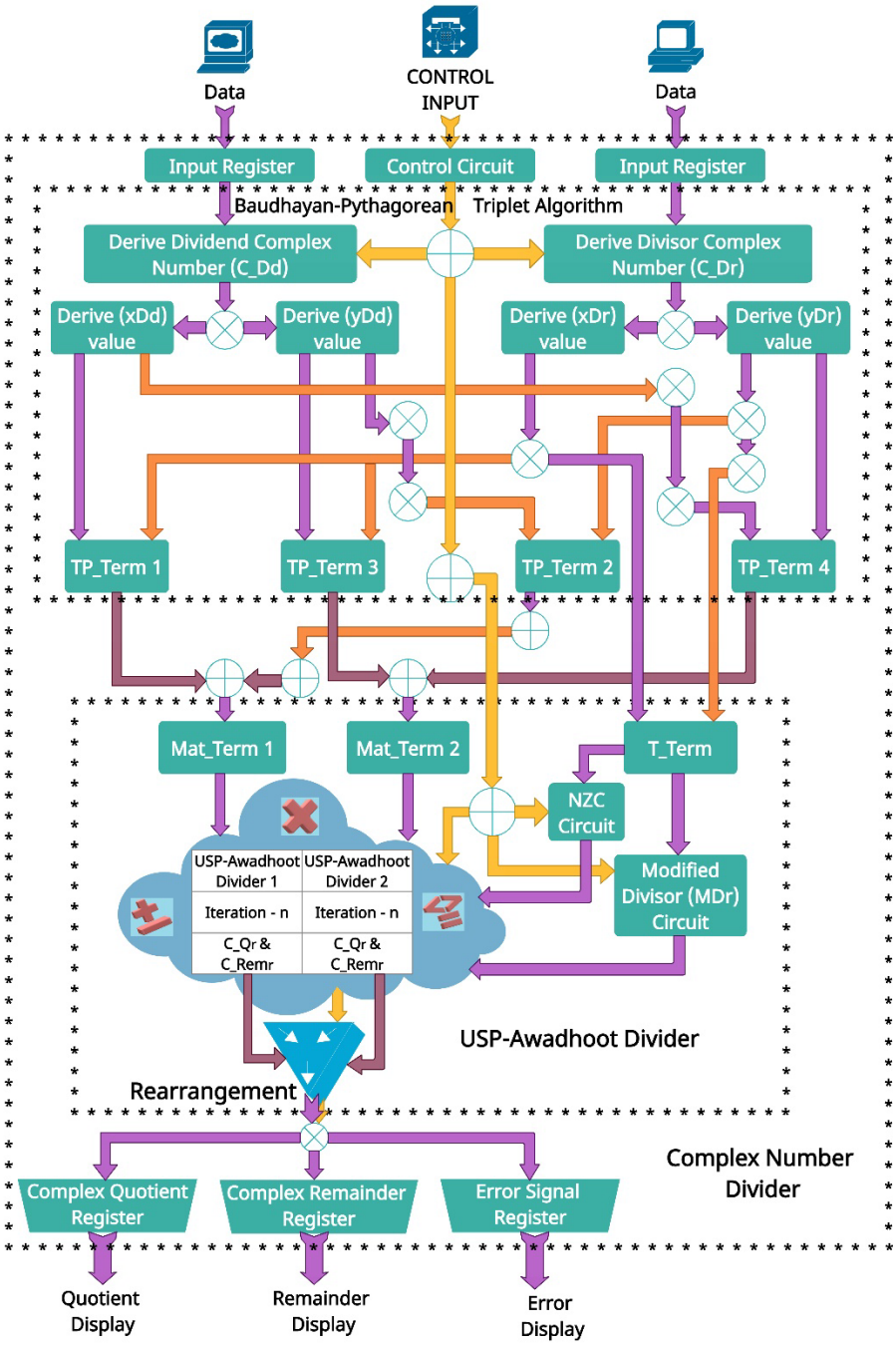


Figure 8. Schematic block diagram of the complex divider.

A detailed list of the essential terms associated with the complex divider are given as:

- Complex number one is termed as $r_1 = x_1 + y_1 i$.
- Complex number two is termed as $r_2 = x_2 + y_2 i$.
- Dividend complex number is termed as "C_D_d".
- Divisor complex number is termed as "C_D_r".
- The real number coefficient of the dividend complex number is termed as "xD_d".
- The imaginary number coefficient of the dividend complex number is termed as "yD_d".
- The real number coefficient of the divisor complex number is termed as "xD_r".
- The imaginary number coefficient of the divisor complex number is termed as "yD_r".
- The first triplet product term is named as "TP_Term1".
- The second triplet product term is named as "TP_Term2".
- The third triplet product term is named as "TP_Term3".
- The fourth triplet product term is named as "TP_Term4".
- The triplet term is named as "T_Term".
- The first triplet matrix term is named as "Mat_Term1".
- The second triplet matrix term is named as "Mat_Term2".
- The USP-Awadhoot Dividend complex number is termed as "C_D_{d1}".
- The USP-Awadhoot Divisor complex number is termed as "C_D_{r1}".
- The real number coefficient of the USP-Awadhoot quotient is termed as "C_Q_r".
- The real number coefficient of the USP-Awadhoot remainder is termed as "C_Rem_r".
- The Imaginary number coefficient of the USP-Awadhoot quotient and termed as "C_Q_i".
- The Imaginary number coefficient of the USP-Awadhoot remainder and termed as "C_Rem_i".
- The final quotient complex number is termed as "C_Q".
- The final remainder complex number is termed as "C_Rem".

Figure 8 illustrates the process of complex division implementation based on the Baudhayan-Pythagorean triplet algorithm and the proposed novel USP-Awadhoot divider circuit block. The implementation consists of three sections: the Baudhayan-Pythagorean triplet algorithm circuit block, the novel USP-Awadhoot divider circuit block, and the complex number re-arrangement circuit block. Unlike Smith and Stewart's algorithm, which provides an additional pre-scaling by a factor of y_2 and x_2 [46], as expressed in equations (26) and (27), the initial part of the proposed complex division is similar to the generalized formula but it is interpreted differently, in terms of the Baudhayan-Pythagoras triplet form (as expressed in equations (21) to (25)).

$$\frac{r_1}{r_2} = \frac{x_1 + iy_1}{x_2 + iy_2} = \frac{\frac{y_2}{x_2}(x_1 + y_1)}{\frac{y_2}{x_2}(x_2 + y_2)} + i\frac{\frac{y_2}{x_2}(y_1 - x_1)}{\frac{y_2}{x_2}(x_2 + y_2)} \qquad if\ \ (x_2 \geq y_2) \qquad (26)$$

$$\frac{r_1}{r_2} = \frac{x_1 + iy_1}{x_2 + iy_2} = \frac{\frac{x_2}{y_2}(x_1 + y_1)}{\frac{x_2}{y_2}(x_2 + y_2)} + i\frac{\frac{x_2}{y_2}(x_1 - y_1)}{\frac{x_2}{y_2}(x_2 + y_2)} \qquad if\ \ (x_2 \leq y_2) \qquad (27)$$

Despite the addition of pre-scaling to enhance robustness, the implementation of Smith and Stewart's algorithm for complex numbers requires a significantly larger area. Smith and Stewart's algorithm does not always ensure the precise rearrangement of the real and imaginary components in the resulting complex quotient and the remainder. This issue arises from the underflow and overflow conditions caused by a larger difference between the divisor and dividend [46, 163]. The need for additional full-width dividers to compute the different scaling factors, underflow, and overflow conditions is a significant drawback of the Smith and Stewart's complex divider implementation. I used the Baudhayan-Pythagorean triplet algorithm as an input circuit stage of the complex divider. Thus, the first stage of the Baudhayan-Pythagorean triplet algorithm circuit block of the proposed divider separates the real and imaginary parts of the input operands. It calculates the intermediate terms by processing the input operands and provides the Mat_Term1, Mat_Term2, and T_Term values to the proposed USP-Awadhoot divider in the next stage. The Mat_Term1, Mat_Term2, and T_Term values are essential for keeping the operations in bounded conditions and to avoid underflow and overflow by using the proposed novel USP-Awadhoot divider, which works on reducing the distance between the divisor and the dividend. During the second stage, the USP-Awadhoot divider circuit block generates two sets of the quotient and remainder values/signals separately as an output. In the final stage of the proposed complex divider, the complex number re-arrangement circuit rearranges the real and imaginary parts of the quotient and the remainder of the complex number. It provides calculated quantities for other displays, storage, or further communication.

4.2 Circuit illustration and state diagram

Figure 9 shows the state diagram of the proposed Baudhayan-Pythagorean triplet algorithm, illustrating the circuit's logic flow in the first part of the complex divider circuit implementation. Figure 10 illustrates the FPGA circuit implementation of the proposed Baudhayan-Pythagorean triplet algorithm according to the complex divider's schematic block diagram and logic flow state diagram.

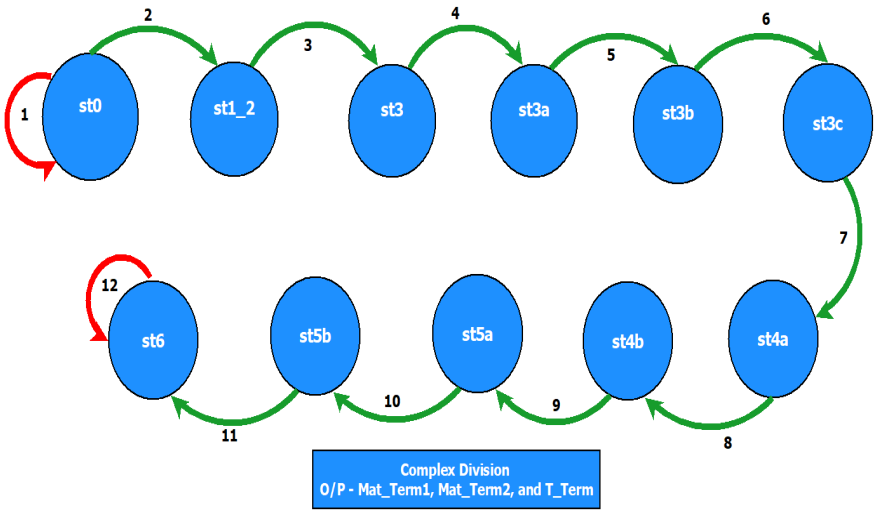


Figure 9. State diagram of the proposed Baudhayan-Pythagorean Triplet algorithm.

The State diagram of the proposed Baudhayan-Pythagorean triplet algorithm indicates the present and next stage conditions depending on the situation of the input operands and control signals. The complex divider circuit's input operands include the dividend, divisor, reset, clock, and cd_enable signals. The FSM stages for the proposed circuit comprise the following states:

- **st0:** This is the initial state. This state takes the input values and transfers them to registers. It waits until cd_enable is one and reset (RST) is 0 to pass to the next state (st1_2); else next state is st0.
- **st1_2:** If the clock and cd_enable signals are applied, and the reset (RST) signal is low, this stage converts the standard logic vectors $x_1, x_2, y_1, and\ y_2$ received from the input operands to integers xD_d, yD_d, xD_r, and yD_r, respectively. If the reset (RST) signal is still at zero, the next state is st3.
- **st3:** If the clock and cd_enable signals are applied, and the reset (RST) signal is low, it calculates TP_term1. If the reset (RST) signal is still at zero, the next state is st5a.
- **st3a:** If the clock and cd_enable signals are applied, and the reset (RST) signal is low, it calculates TP_term2. If the reset (RST) signal is still at zero, the next state is st3a.
- **st3b:** If the clock and cd_enable signals are applied, and the reset (RST) signal is low, it calculates TP_term3. If the reset (RST) signal is still at zero, the next state is st3c.
- **st3c:** If the clock and cd_enable signals are applied, and the reset (RST) signal is low, it calculates TP_term4. If the reset (RST) signal is still at zero, the next state is st4a.
- **st4a:** If the clock and cd_enable signals are applied, and the reset (RST) signal is low, it partially calculates T_term. If the reset (RST) signal is still at zero, the next state is st4b.
- **st4b:** If the clock and cd_enable signals are applied, and the reset (RST) signal is low, it fully calculates T_term. If the reset (RST) signal is still at zero, the next state is st5.
- **st5:** If the clock and cd_enable signals are applied, and the reset (RST) signal is low, it calculates Mat_term1. If the reset (RST) signal is still at zero, the next state is st5b.
- **st5b:** If the clock and cd_enable signals are applied, and the reset (RST) signal is low, it calculates Mat_term2. If the reset (RST) signal is still at zero, the next state is st6.
- **st6:** The final state, used to indicate, hold and transfer the results from the calculations to the outputs to connect with the USP-Awadhoot divider as a second part of the complex divider.

The schematic diagram of the proposed circuit implementation of the Baudhayan-Pythagorean triplet algorithm is shown in Figure 10. For the sake of easier understanding, the implementation of the proposed Baudhayan-Pythagorean algorithm is illustrated by three subsequent stages: input, intermediate, and output stages (see Figure 10). The different signals used in the circuit implementation of the Baudhayan-Pythagoras triplet algorithm are grouped into input operand, control, output, and indicator signal groups.

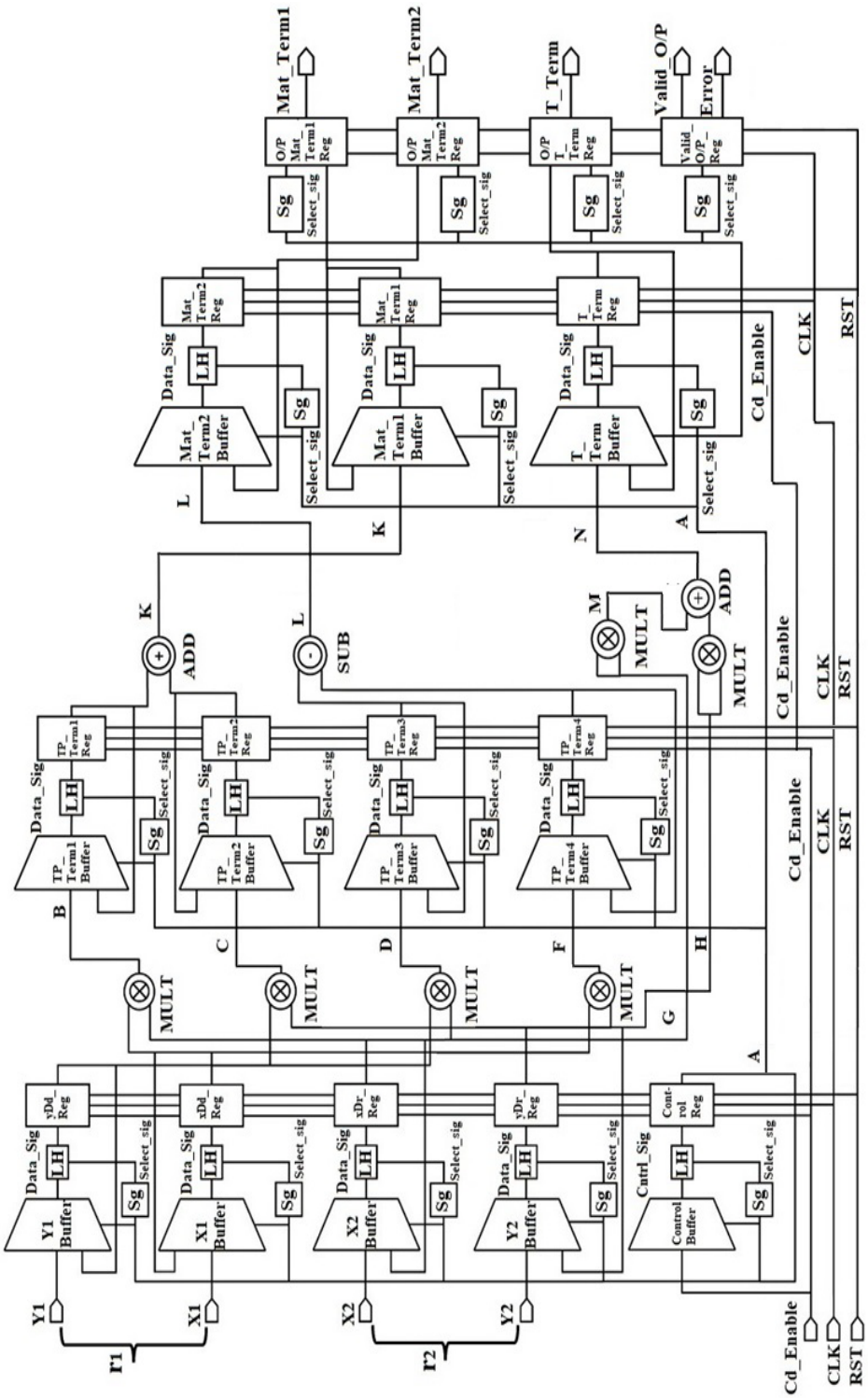


Figure 10. Schematic diagram of the proposed Baudhayan-Pythagorean triplet section of the complex divider.

All signals are divided into groups such as input operand data signals (r_1, r_2), computation completion acknowledgment (*Valid_O/P*), and error (Error) indicator signals. Enable (cd_enable), clock (CLK), and reset (RST) are considered to be control signals, and Mat_term1, Mat_term2, and T_term are output signals. The control group clock signal (CLK) provides the timing reference signal for the computation and the reference clock signal's period value depends on the operating frequency of the circuit. When the CLK signal continues generating the reference signal and the control group signals (cd_enable and RST), both possess low logic values; the operation of the circuit is then in an idle state. The input operand, output, and indicator group signal values are in a high-impedance tri-state condition during the idle state.

As shown in Figure 10, the input operand signals $r_1\ and\ r_2$ provide two complex numbers, which are used to perform division operations based on the current states of the cd_enable, CLK, and RST control signals. In the input circuit stage, all real and imaginary parts of the input operands are separated and stored in the input buffer and wait until the cd_enable signal is high (1) and reset (RST) is low (0). The output circuit stage initializes the Mat_Term1, Mat_Term2, and T_Term signals from the output and indicator signal group to 00, assuring that the previous computation results are not involved in the current computation. Once the cd_enable signal is applied, this signal is used to develop a select signal and is stored in the control register to connect with further circuit stages. The input operand data is provided for further computation in the input circuit stage. The input operand data gets stored into $x_1, x_2, y_1\ and\ y_2$ buffers respectively to extract $xD_d,\ yD_d, xD_r, and\ yD_r$ values for the generation of B to G signals.

The intermediate circuit stage receives the signal B to signal G data from the input circuit stage. The forward signals B, C, D, F, H, and G are generated from the TP_Term1 to TP_Term4 computation. The computed data is stored in separate buffers and made available for further computation, based on the select signal data required to calculate partial Mat_Term1, partial Mat_Term2, and partial T_Term values. Signals K, L, and N indicate the partial Mat_Term1, partial Mat_Term2, and partial T_Term values and transfer respective data to the next circuit stage.

The output circuit stage receives signals K, L, and N from the intermediate circuit stage and stores the respective data in Mat_Term1, Mat_Term2, and T_Term buffers. The output circuit initializes the indicator signals for computation completion acknowledgment (*Valid_O/P*) and error (Error) to a value of 00, to ensure that no residual data from previous computations is included. When the reset (RST) signal is deactivated, and the cd_enable signal is activated during the initial state, the partial values are further converted into the final Mat_Term1, Mat_Term2, and T_Term values, based on the selected signal logic, and further utilized as Mat_term1, Mat_term2, and T_term output group signals. These signals are further connected with the USP-Awadhoot divider circuit block and complex number re-arrangement circuit, to receive the final division results of complex input operands. The *Valid_O/P* and Error signals indicate computation completion and invalid operating conditions, respectively. If a logic high signal activates a reset (RST) signal, the divider circuit suspends its current state of computation operation and resets itself to the initial state. After completing the computation operation, depending on the completion of data computation, *Valid_O/P* and error (Error) are updated, and validating the computation and O/P results, i.e., whether values are correct or incorrect.

4.3 Summary

The basic steps involved in the proposed complex divider can be summarized as:

Step 1 - Define the dividend complex number (C_D_d) and Divisor complex number (C_D_r) from given complex numbers $r_1 = x_1 + y_1 i$ and $r_2 = x_2 + y_2 i$; where C_D_d = r_1 and C_D_r = r_2.

Step 2 - Derive Cartesian coordinates: $xD_d,\ yD_d, xD_r, and\ yD_r$.

$xD_d = x_1$
$yD_d = y_1$
$xD_r = x_2$
$yD_r = y_2$

Step 3 - Derive the triplet product term (TP_Term) values.

TP_Term1 = ($xD_d \times xD_r$)
TP_Term2 = ($yD_d \times yD_r$)
TP_Term3 = ($xD_r \times yD_d$)
TP_Term4 = ($xD_d \times yD_r$)

Step 4 - Derive the triplet term (T_Term) value.

T_Term = $(xD_r)^2$ + $(yD_r)^2$

Step 5 - Derive the triplet matrix term (Mat_Term) value.

Mat_Term1 = TP_Term1 + TP_Term2
Mat_Term2 = TP_Term3 - TP_Term4

Step 6 - Supply Mat_Term1, Mat_Term2, and T_Term values to the USP-Awadhoot divider for the final division sub-process.

Step 7 - USP-Awadhoot divider one receives Mat_Term1 and T_Term values to perform the division sub-process.

C_D_{d1} = Mat_Term1
C_D_{r1} = T_Term

Giving the computation results C_Q_r and C_Rem_r

Step 8 - USP-Awadhoot divider two receives Mat_Term2 and T_Term values to perform the division sub-process.

C_D_{d2} = Mat_Term2
C_D_{r2} = T_Term

Giving the computation results C_Q_i and C_Rem_i

Step 9 – Concatenate the step 7 and step 8 computational results to restructure the Cartesian coordinates to get the final quotient and remainder of the complex division.

Quotient = C_Q = (C_Q_r concatenate C_Q_i) and
Remainder = C_Rem = ($\_Rem_r$ concatenate C_Rem_i)

4.4 Chapter conclusion

A detailed explanation of the basic working concept of the proposed Baudhayan-Pythagoras triplet algorithm, used in association with the proposed novel USP-Awadhoot divider, is described in this chapter. The proposed Baudhayan-Pythagorean triplet algorithm helps to simplify inputs, ensuring the separation of the real and imaginary parts of complex numbers for further calculation. This chapter covers RO3 – "Divider algorithm formulation to reduce the criticality of conversion logic". Publication IV covers detailed information on the proposed Baudhayan-Pythagoras triplet algorithm associated with the novel USP-Awadhoot divider.

5 Implementation and performance statistics

This chapter is based on publications I, II, and IV. Very-large-scale integration (VLSI) plays a critical role in the integration of millions of transistors onto a single chip, providing the foundation for today's cutting-edge devices. A crucial aspect of the VLSI design flow is the use of hardware description languages (HDLs) such as VHDL, which serve as the cornerstone for defining and simulating hardware functionality during the front-end design process. The front end involves creating high-level abstractions of system behavior, requiring precise algorithms to ensure logical correctness, optimized performance, and scalability. Conversely, the back-end process focuses on translating these abstractions into physical realizations, including circuit layout, synthesis, and the detailed specifications necessary for fabrication. Algorithms bridge these domains by serving as an intermediate representation that facilitates the logical-to-physical transformation, enabling the efficient and reliable implementation of the hardware. As previously mentioned, cost, area, execution time, and energy consumption are the essential evaluation metrics for any divider implementation. In some applications, such as consumer electronics and embedded systems, the focus is often on minimizing the area and cost to achieve compact, affordable designs. In contrast, high-performance applications, such as biomedical computations, prioritize low latency and fast execution times to ensure timely and accurate results.

Implementation and performance statistics can be divided into two essential parts: implementation area and latency time analysis. I implemented the synthesizable architecture of the proposed divider to analyze its functionality and verify the correctness of its calculation logic. This approach also helped determine the resources required to deploy the proposed divider on an FPGA and enabled a comparison with various other divider implementations. The behavioral simulation results were compared with standard theoretical calculations, to verify the correctness of the proposed divider algorithm. Additionally, timing/waveform analysis confirms the execution time/latency of the divider implementation. This analysis is especially critical for data-dependent divider implementations, where the operand values directly influence latency.

Figure 11 provides a generalized architectural illustration of FPGA building blocks. Configurable logic blocks (CLB) are the core building blocks of an FPGA. The architecture of CLBs is crucial for the FPGA's ability to implement various digital logic functions. While the exact architecture varies across FPGA families from different manufacturers, most CLBs share standard configuration features. Considering Figure 11, one CLB consists of two logic slices, and each slice contains four look-up tables (LUT), four flip-flops, latches, and two multiplexers (F7MUX and F8MUX). A LUT is a collection of hard-wired logic gates on an FPGA. LUTs store a predefined list of outputs for every possible combination of inputs and provide a fast way to retrieve the output of a logic operation. A flip-flop is a memory circuit which is capable of two stable states with a single bit. A multiplexer, or 'mux', is a circuit that selects between two or more inputs and outputs the selected input. Different FPGA families implement slices and LUTs differently. For example, a logic slice on a Virtex-II FPGA has two LUTs and two flip-flops; whereas, a logic slice on a Virtex-5 FPGA has four LUTs and four flip-flops. In an FPGA, a fixed number of identical transistors (N_r) is required to build each LUT, flip-flop, and multiplexer. Thus, knowing the number of each LUT, flip-flop, and multiplexer used in implementing a particular circuitry on an FPGA ultimately leads to a resistor-transistor circuit level, giving us the final count of required transistors.

FPGAs (e.g., the Xilinx XCZU7EV-FFVC1156-2-E) consist of millions of programmable logic gates (which are constructed from transistors) but are described at a much higher level of abstraction like CLBs, the number of slices, LUTs, FFs, and multiplexers. It provides an accurate estimate of the required transistor number and a precise value of the implementation area, as the transistor area is fixed. The datasheet of the Xilinx XCZU7EV-FFVC1156-2-E FPGA states that it has approximately 230,400 LUTs and 460,800 FFs. In the UltraScale+ architecture, including the XCZU7EV, the LUTs are typically 6-input LUTs (LUT6), which can be stored using 64 SRAM cells and accessed by a 64:1 multiplexer. Generally, six transistors are needed for one SRAM cell and two transistors are needed to build a 2:1 multiplexer; thus, approximately 384 (for SRAM) + 126 (for MUX) = 510 transistors are needed to build one 6-input LUT.

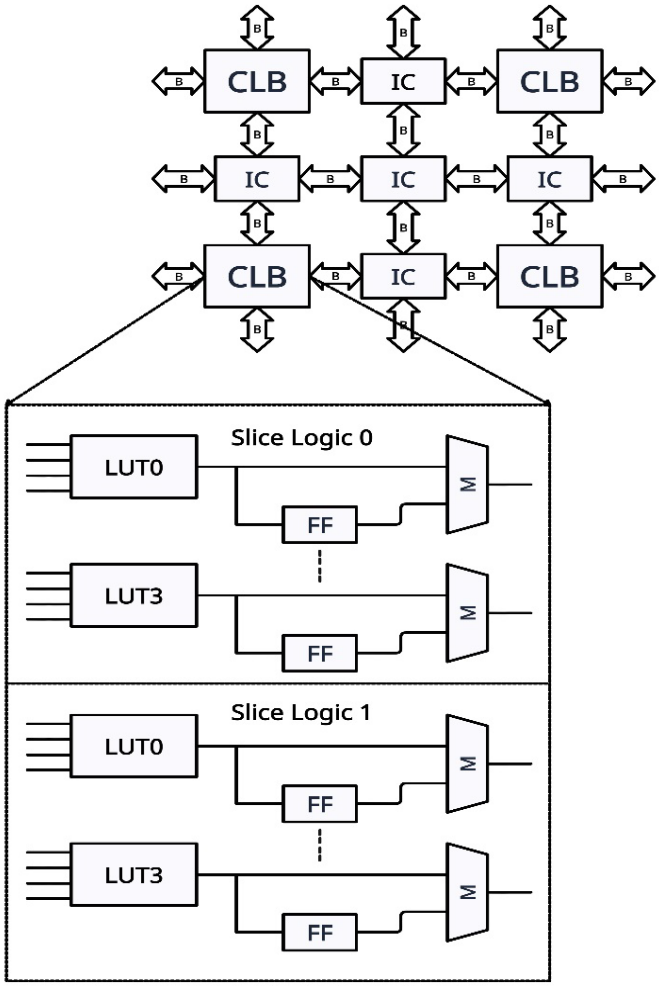


Figure 11. Generalized architectural illustration of FPGA building blocks.

Similarly, in the Xilinx UltraScale+ architecture, including the XCZU7EV-FFVC1156-2-E, each slice contains flip-flops (registers) used for storage and sequential logic [172, 173]. The flip-flops in these slices are often similar in design to conventional D flip-flops,

requiring around 20 transistors per flip-flop [172, 173]. This number gives a rough estimate, and the actual count might vary slightly, depending on specific optimization in the CLB slice implementation within the UltraScale+ architecture. In the chosen FPGA family, LUTs usually have two to six inputs. A register is a group of flip-flops used to store a bit pattern. A register on an FPGA has a clock, input data, output data, and enabled signal ports. Logic slices, look-up tables (LUTs), flip-flops (FFs), and multiplexers are fundamental resources in FPGA architectures. Their utilization provides an effective basis for evaluating and comparing the divider circuit designs.

5.1 Implementation and performance analysis of the USP-Awadhoot divider circuit block

The proposed novel USP-Awadhoot divider is implemented in VHDL (Very high-speed integrated circuits Hardware Description Language). In order to realize the theoretical concept and idea of the proposed novel USP-Awadhoot divider, we developed a synthesizable architecture, which is also referred to as the USP-Awadhoot divider circuit block. This synthesizable implementation provides a unified way of comparing and testing the divider. To implement and test the proposed USP-Awadhoot divider, we used two vendor architectures to cross-verify the simulation results by comparing the outputs separately with the truth table.

1. Vivado 2016 simulation tool with the Zybo development board based on Xilinx[1] Zynq XC7Z010, XCZU7EV-FFVC1156-2-E with Zynq UltraScale + MPSoC.
2. Quartus Prime Lite simulation software with the Cyclone IV development board based on the EP4CE6E22C8N Cyclone IV FPGA manufactured by Altera[2].

Here, two different FPGAs (Xilinx and Altera) were used to test the correctness of the logical results when implemented with differently structured FPGAs. Unless otherwise specified, the Xilinx implementation and simulation statistics of the proposed divider are considered further for comparisons, as most available data for the applications used Xilinx FPGA to test and implement various dividers.

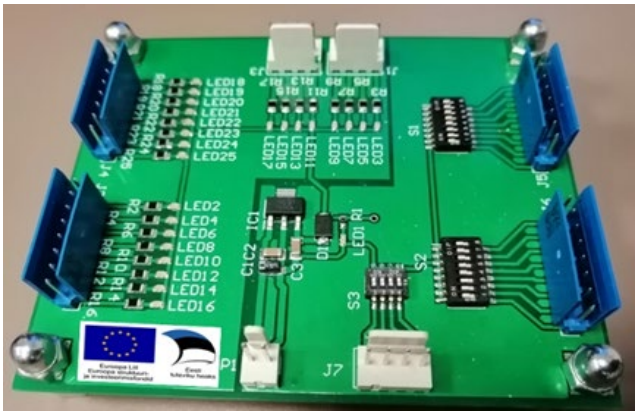


Figure 12. Logic test bench board.

A truth table, covering all possible combinations of the operands comprising the valid or theoretical results, was referred to in order to check the validity of the generated output.

[1] AMD acquired Xilinx in 2022.
[2] Intel acquired Altera in 2015.

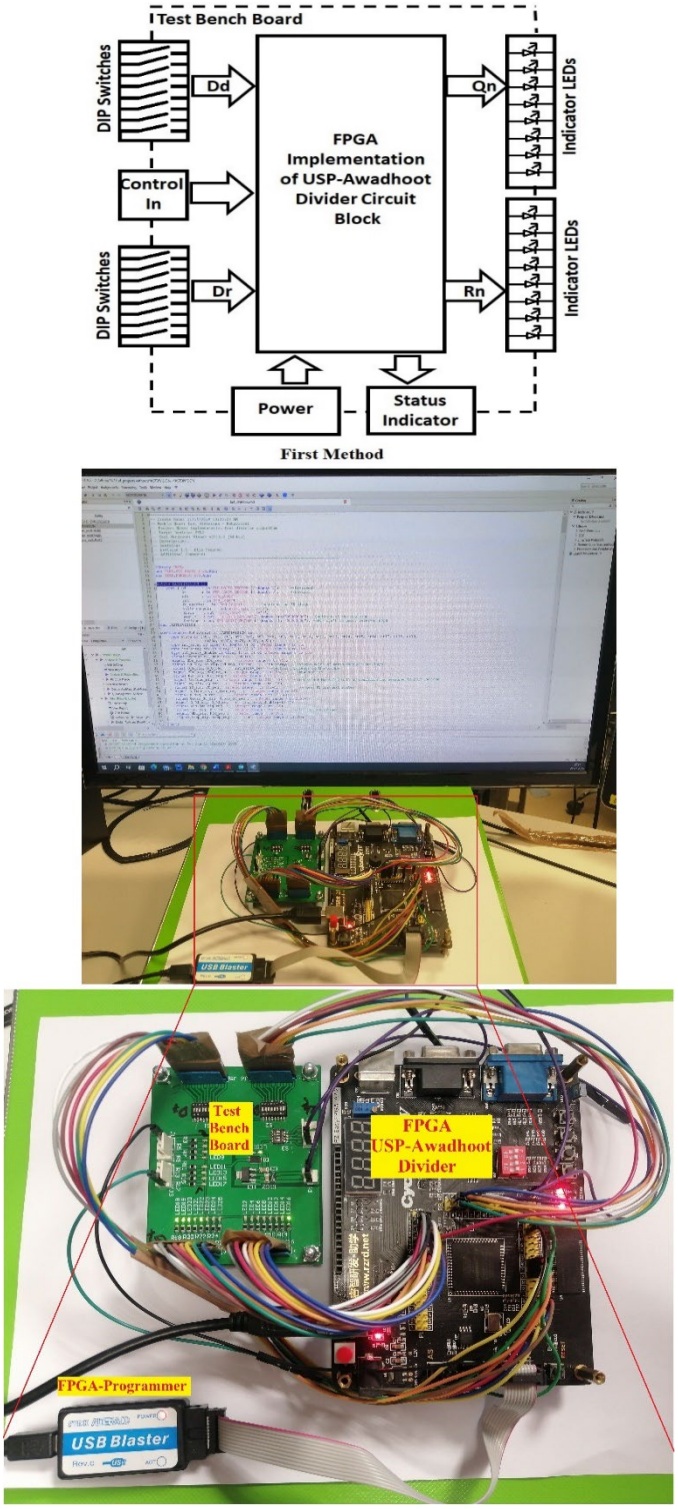


Figure 13. Test arrangements for the first method.

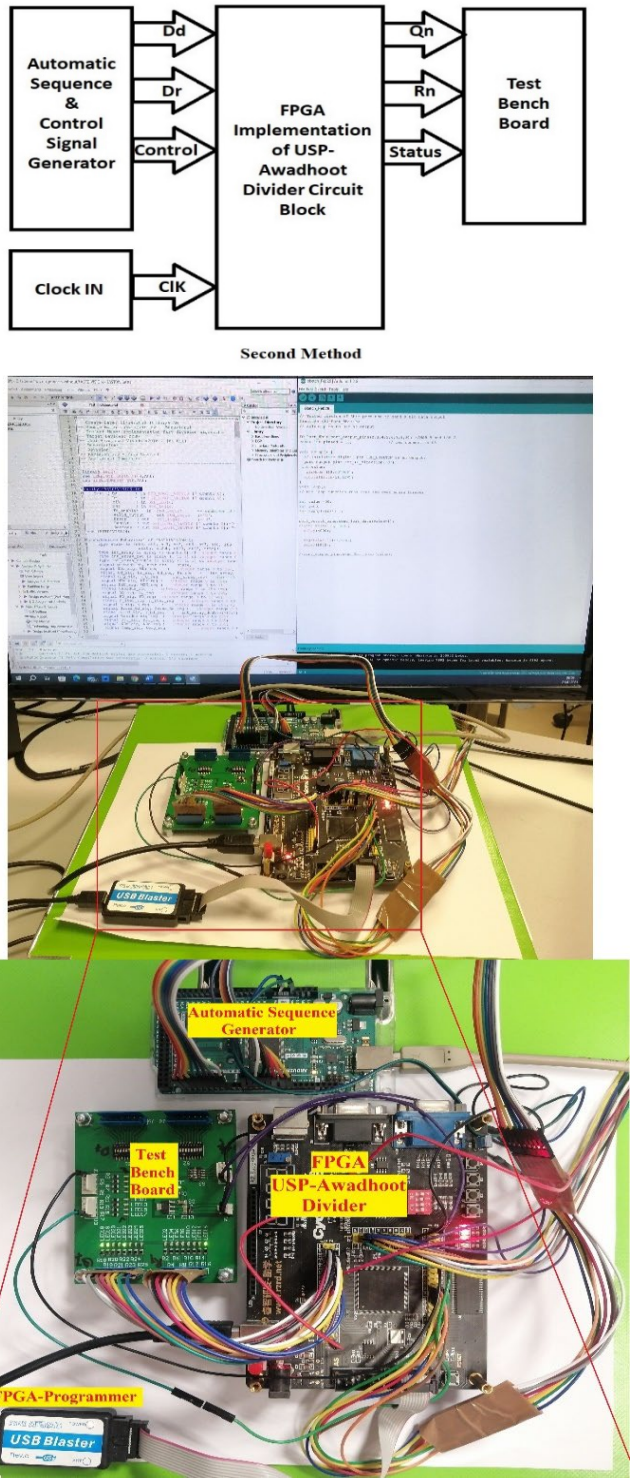


Figure 14. Test arrangements for the second method.

After connecting the test bench board to the FPGA test board, the proposed divider was tested with all possible input operand combinations. Operand values are selected from the test bench and provided to the divider's input ports. Control signals generated by the test bench trigger the divider to perform the division operation, producing the quotient and remainder. These outputs are then sent back to the test bench board for displaying. Finally, the results are compared against the truth table to verify the accuracy of the divider results. Both experimental FPGA test boards were used to verify all possible input operand conditions. During verification, the input operand values were selected in both sequential and random orders. A random number generator (RNG) and a sequential number generator (SNG) were used to evaluate the random and sequential operation of the proposed divider implementation. As shown in Figure 12, I designed and used a logic test bench board which was capable of providing input and output operands with varying word sizes, to test the proposed implementation. The FPGA test board is connected to the test bench board through connectors, which supply the input operands and display the generated quotient and remainder outputs on LEDs. We tested the operation of the proposed novel USP-Awadhoot divider in multiple ways.

As illustrated in Figure 13, the first method involves utilizing the test bench board's dual in-line package (DIP) switches as input operands and displaying output through indicator LEDs. This configuration is created to manually test the sequential and random operation of the proposed divider. Figure 14, shows that the second technique, the automatic sequence generator (sequential and random), and the controlling signal are connected to the input operands of the proposed divider built on the development boards, and the output operands are connected to the test bench board for result verification. This configuration is meant to automatically test the sequential and random operation of the proposed division. An automatic sequence generator (sequential and random) and the controlling signal are created using Arduino or other embedded systems; an external function generator provides a working clock signal. In the third method, we used simulation tools (Vivado and Quartus Prime Lite) to verify the sequential and random operation of the proposed divider. Finally, the output accuracy is validated by comparing the results provided by three verification methods to the theoretical results truth table.

After verifying the simulations and hardware implementation of different versions of the proposed divider circuit, the implementation statistics of every version are mapped to the LUTs, flip-flop registers, multiplexers, latches, basic gates, and clock frequency. It specifies the number of transistors or gates used, indicating the quantity of implemented area or hardware resources used in the proposed novel USP-Awadhoot divider. As explained previously in Chapter 2, all processes involved in the pre-processing, processing, and post-processing circuit stages are implemented sequentially. Bounded input-outputs (I/Os) are divided into two data operands: input and output data lines. Input control lines are used to control the divider's operation, and output status lines are used to report an error if it occurs during computation.

The pre-processing circuit allows data from input data lines, i.e., dividend (D_d) and divisor (D_r), to be stored in input registers. The most significant bits (MSB) are stored as a separate hexadecimal integer number. The least significant bits (LSB) are stored as another hexadecimal integer in an array of hexadecimal integer elements for the dividend. The same process is applied to both the dividend (D_d), and the divisor (D_r). Concurrently, the pre-processing circuit stage formulates the *FD* and ND_r values by only working on the least significant hexadecimal part of the divisor.

The processing circuit stage concurrently executes the ND_d and GQ_n calculation step for each group dividend (GD_d) and utilizes the pre-defined values for error conditions. In the last group dividend (GD_d), the processing circuit stage concurrently executes the residue/remainder and additional quotient calculations if needed. The processing circuit stage introduces an extra buffer and counters to improve the expected group residue/remainder and additional quotient calculations. Once the last group dividend (GD_d) calculations are completed, the condition selection and rearrangement circuits are activated in the post-processing circuit stage to compute the final values for quotient and residue as per the display/storage requirements.

Figures 15-17 show the resource requirements, power, and frequency estimations of multiple versions of the proposed USP-Awadhoot divider implementations; each implementation is referred to as a 'version'. Version V8.1 is an 8-bit operand implementation of the proposed USP-Awadhoot divider, while versions V16.1, V24.1, and V31.1 are the 16-bit operand, 24-bit operand, and 31-bit operand implementations, respectively, based on the V8.1 version. The V31.1 implementation uses 31-bit operand due to software configuration limitations to avoid overflow conditions during conversion.

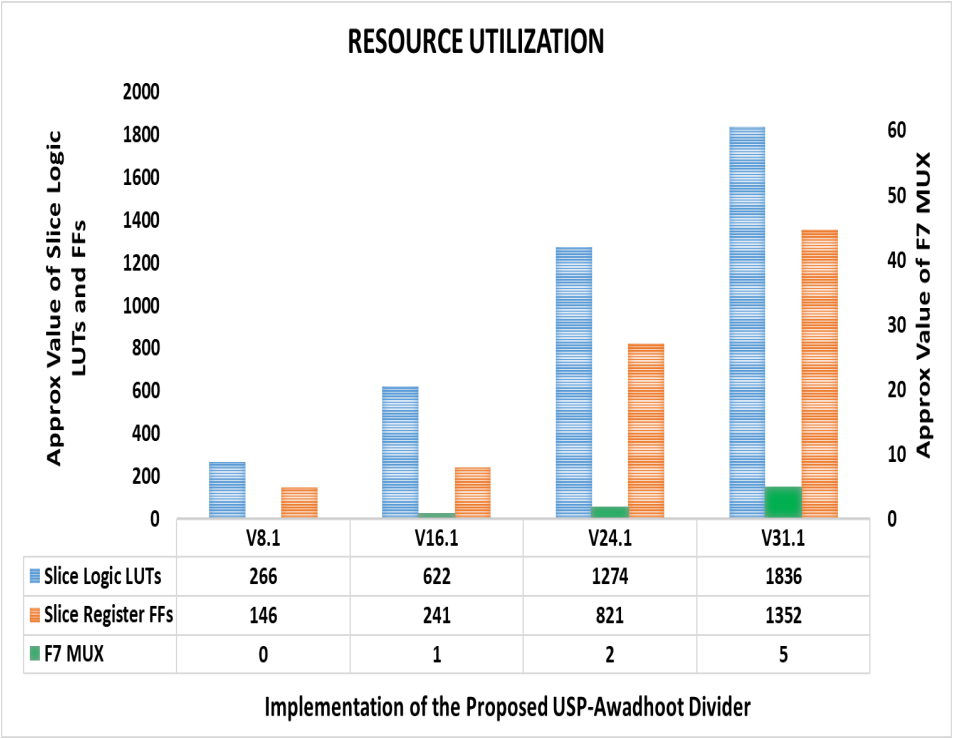


Figure 15. Hardware resource utilization.

Figures 15-17 illustrate the actual data for the proposed circuit implementation based on the Xilinx FPGA simulation tool. This is used as a baseline for comparing the numerous alternatives and enables the drawing of a comparative analysis, as in Sections 5.3 and 5.4.

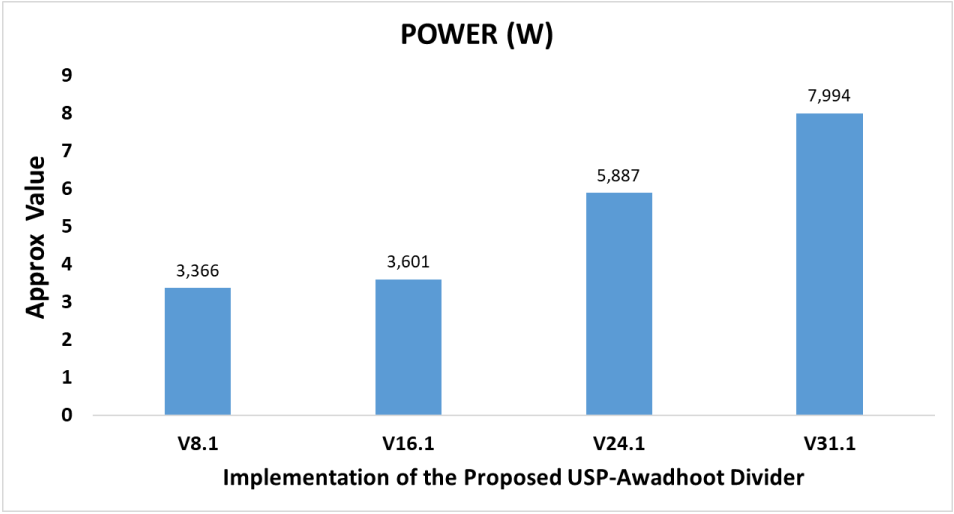


Figure 16. Estimated power consumption.

A study of different implementation versions indicates the total trade-off between area, time, and power estimation. So, depending on the application, one must decide which implementation version must be utilized. As per Figures 15-17, the V8.1 version, which represents the 8-bit implementation of the proposed divider, requires 266 slice logic LUTs, 146 slice register flip-flops, and 37 bounded input-outputs with zero latches, DSPs, or eight-input multiplexers. It utilized 0.12% of the available LUTs and 0.03% of the available slice register flip-flops, in total. Simulation confirms that it operates at a moderate divider clock frequency of up to 285 MHz and consumes 3.366 watts of estimated power.

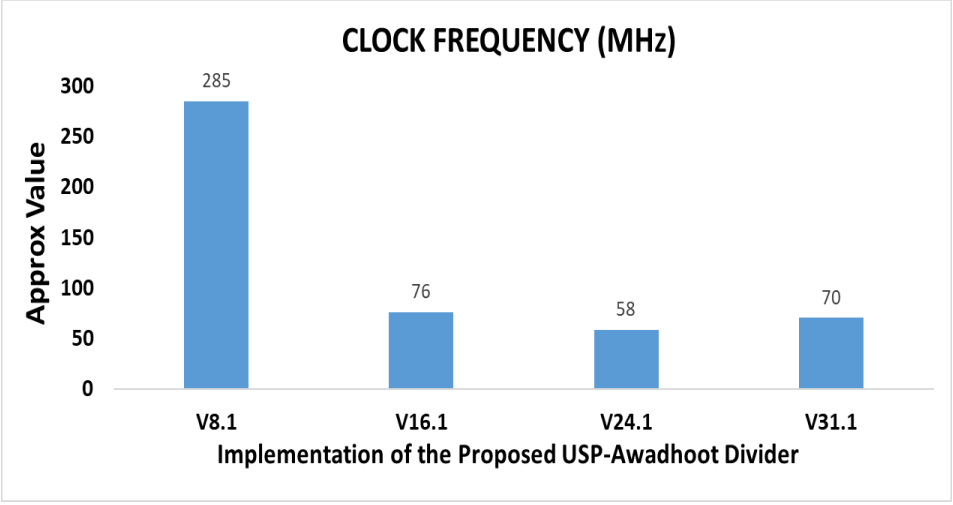


Figure 17. Divider clock frequency.

As the bit size increases, there is a progressive increase in resource usage, which is nearly double that of the prior version. The V16.1 version, which represents the 16-bit implementation of the proposed divider, requires 622 slice logic LUTs and 241 slice register flip-flops. Simulation confirms that it operates at 76 MHz clock frequency and consumes 3.601 watts of estimated power. It utilized 0.26% of the available LUTs and 0.05% of the available slice register flip-flops in total. Similarly, the V24.1 and V31.1 versions of the proposed divider, which represent 24-bit and 31-bit implementations, require 1274 and 1836 slice logic LUTs, as well as 821 and 1352 slice register flip-flops, respectively. It utilized 0.55% to 0.79% of the available LUTs and 0.17% to 0.29% of the available slice register flip-flops in total. Furthermore, we used versions V8.1 and V16.1 of the proposed divider circuit in all of the comparative implementation resource utilization studies.

The divider uses the divider clock frequency as a reference for performing division operations. The proposed USP-Awadhoot divider's behavior is mapped with a latency time performance function, designed to keep track of the number of clock cycles required for a given pair of input operands. So, the latency analysis is performed in terms of clock cycles. Simulations identify the performance of the proposed divider latency time by analyzing the clock cycle calculation of the best and worst conditions. We consider two ways to evaluate the proposed divider's latency time performance. The first is a sequential truth table, which ensures that each combination of input operands is considered during execution and its associated data is saved. The second option, RNG, is suitable for evaluating operands with larger word sizes. The execution time of the data-dependent divider is decided by how far the divisor is from the dividend rather than by the value of the dividend. The greater the distance between the dividend and the divisor, the longer the execution time.

Variable latency can be used to provide a variable conversion rate or time. Achieving variable latency in the divider is critical, due to the difficulty of synchronizing iterations to get the correct results. To understand the nature of the variable latency in the proposed divider, I performed a clock performance analysis of the entire range of 8-bit operands using the RNG and the sequential truth table, with 65K possible combinations. The RNG method is only considered for larger bit sizes, due to the possibility of billions of combinations. The clock performance analysis of the proposed USP-Awadhoot divider is performed at a 125MHz clock frequency.

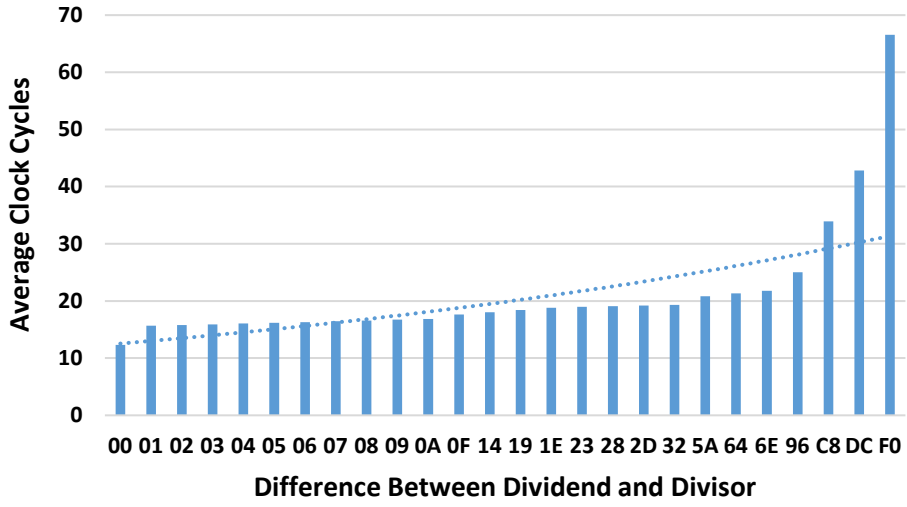


Figure 18. Clock performance analysis based on the distance between the dividend and the divisor.

Figure 18 illustrates the proposed divider's behavior based on the difference between the dividend and divisor values. It demonstrates that the proposed divider circuit uses the fewest clock cycles (approximately 1 to 3) when the divisor value is 0, as division by zero results in an invalid condition. When the distance between the divisor and dividend values is small, the minimum necessary average number of clock cycles remains in the range of thirteen to twenty-four clock cycles. When the divisor value is one, the smallest number of clock cycles is required (7 clock cycles). If the dividend value was previously confirmed to be non-zero, the final quotient value is calculated directly after ensuring that the divisor value is unity. Mid-range operand combinations required fifteen to thirty-five clock cycles, whereas large-range operand combinations required twenty-eight to sixty-eight clock cycles. A dividend value of zero is an exception to the invalid condition; when the input operand value indicates that the dividend and divisor values are both zero, the proposed divider circuit needs slightly longer clock cycles (17 clock cycles) to complete the execution process. The main reason for this is that the circuit first detects the dividend value to identify it as a nonzero value. If a zero dividend value is detected, the temporary output is set to zero and it checks the divisor for a non-zero value. If it detects a non-zero value, the quotient is set to zero; otherwise, the error signal indicates an invalid condition.

5.2 Waveform analysis

The functional waveform analysis of the proposed divider based on the USP-Awadhoot division algorithm is discussed in this section. The operating conditions of the various signals used or generated by implementations in various scenarios, such as idle/initial and off/on states, are presented in Figure 19. We consider multiple dividend and divisor combinations in order to better understand various signals and data during an individual conversion process. The nine signal data were studied using waveform analysis, to provide a clear picture of the proposed divider's working conditions based on the USP-Awadhoot division algorithm. The nine signals required for waveform analysis are:

the reference clock (*CLK*), dividend (D_d), divisor (D_r), enable (*fd_enable*), quotient (*Q_Result*), the remainder (*Rem_Residue*), computation completion acknowledgment (*Valid_O/P*), error (*Error*), and reset (*RST*). These nine signals are divided into five groups based on their nature: the reference group, the I/P operand group, the control group, the O/P results group, and an indicator group.

Different dividend and divisor combinations require different clock cycles for computation; the timing reference signal for computation execution is provided by the reference group CLK signal. The dividend (D_d) and divisor (D_r) signals are part of the I/P operand group. The Control group consists of *fd_enable* and *RST* signals, to provide start and end control signals for the computation process. The indicator group consists of *Valid_O/P* and Error signals, which indicate computation completion and notify the system of invalid working conditions or erroneous execution. The final and most important O/P results group consists of the Q_Result and Rem_Residue signals, which provide the quotient and remainder values as the result of the division operation conducted by the proposed USP-Awadhoot divider.

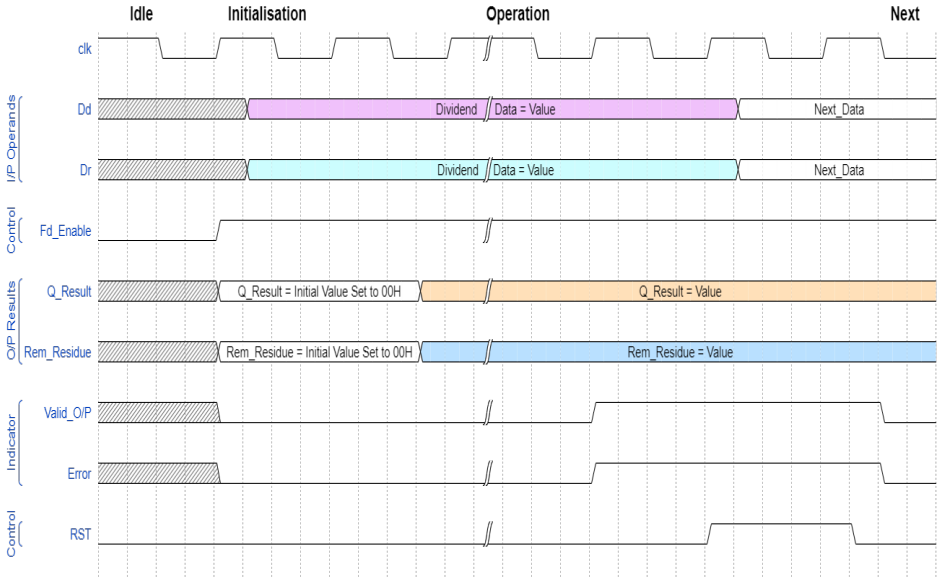


Figure 19. Waveform reference for initial operating condition.

Figure 19 indicates a reference functioning waveform for the initial working conditions. The working waveform is primarily segmented into an idle state, an initialization state, an operation state, and the 'next' state. The idle state shows the proposed divider circuit's non-working or stationary status and the beginning state after a shutdown when only a power source is applied to the circuit. In the idle state, the CLK signal continues to generate the reference signal and the dividend (D_d) and divisor (D_r) signals of the I/P operand group are in the high-impedance tri-state condition. The Control group *fd_enable* and *RST* both have low logic values, suggesting no operation. Similarly, the value of the indicator group and O/P results group's *Q_Result* and *Rem_Residue* signals are in a high impedance tri-state condition, indicating a stationary working condition. After applying the *fd_enable* control signal to the proposed division, the initialization state signals the

next stage. The proposed circuit resets the signal value of the O/P results group to the initial value of 00H during the initialization stage, implying no results at the start. As stated previously, it fetches the dividend (D_d) and divisor (D_r) data from input data lines and stores them in the input operand registers for further computation. The *Valid_O/P* and Error signals in the indicator group are set to logic low values, indicating that no calculation operations have yet been completed. During the operation state, the proposed divider circuit computes the *Q_Result* and the *Rem_Residue* signals. After completing the computing process, the *Valid_O/P* and *Error* signals are modified, and the accuracy of the computation and *quotient* values is validated. The proposed USP-Awadhoot divider suspends its current computing state and resets to its initialization state if a high-logic signal triggers the *RST* signal. The RST signal is set to low logic, signifying an inactive reset signal, allowing the ongoing computing process to compute the final quotient values. Once the quotient value has been determined, the proposed divider is ready to proceed to the next computation, depending on the control group signal.

5.3 Summary of comparative analysis

This section includes a study of the various divider implementations, providing a comprehensive overview of the other implementations that could be used to compare the performance of the proposed divider implementation. It demonstrates the need to establish a good trade-off between time, cost, area, and complexity while selecting a suitable division algorithm for a required application. The Vivado 2016 simulation tool with the Zybo development board based on Xilinx Zynq XC7Z010, XCZU7EV-FFVC1156-2-E with Zynq UltraScale + MPSoC and the Quartus Prime Lite simulation software with the Cyclone IV development board based on the EP4CE6E22C8N Cyclone IV FPGA manufactured by Altera were used to develop and implement the proposed novel USP-Awadhoot divider. The simulation results and FPGA implementation were cross-verified by separately comparing the outputs with the truth table indicating results for all input combinations. Here, two different FPGAs (Xilinx and Altera) were used to test the correctness of the logical results when implemented with different structured FPGAs. Unless otherwise specified, the Xilinx implementation and simulation statistics of the proposed USP-Awadhoot divider are considered further for comparisons, as most applications used Xilinx FPGA to test and implement various dividers.

Restoring and non-restoring algorithms are comprehensive concepts. The restoring algorithm is identical to the actual long division algorithm or the theoretical paper and pencil algorithm, and the non-restoring algorithm is similar to the restoring algorithm except for the restoring stage. These algorithms are the fundamental algorithms of the dividers in the digit recurrence class. Many scholars have investigated the complexity, timing, area, and other properties of the implementations of simple restoring and non-restoring algorithms [1, 3, 66, 68]. Many non-restoring algorithms have been designed and implemented, but the SRT algorithm is the most implemented approach. Many algorithms that appear later are either entirely or partially derived from the digit recurrence concept and functional iteration class division algorithms.

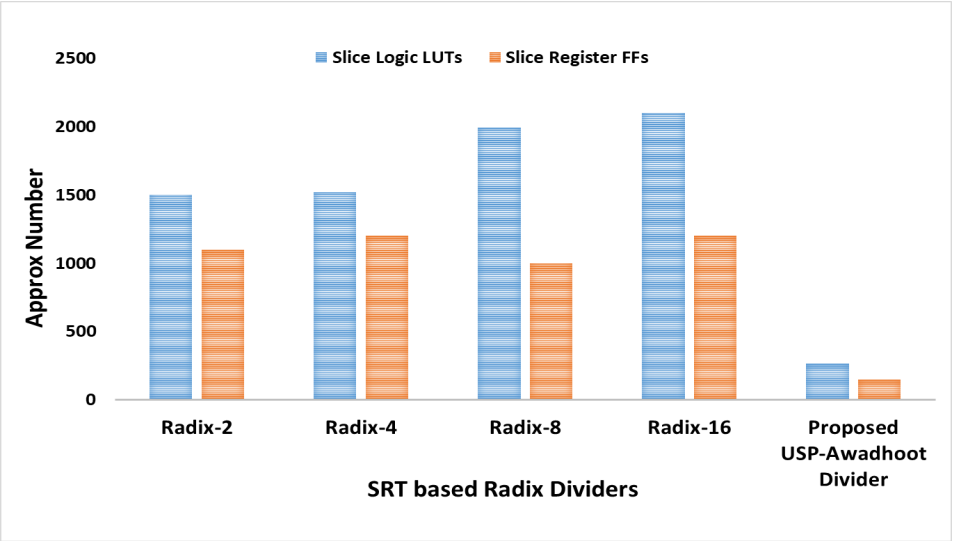


Figure 20. Comparative analysis of the proposed USP-Awadhoot divider with the radix-n based SRT divider.

The basic SRT algorithm was implemented in [5, 9, 12, 17, 19, 21, 35, 51, 58, 68, 76-83] for different applications utilizing different aspects of the algorithm. Figure 20 illustrates the comparative analysis regarding the hardware resource utilization of the proposed USP-Awadhoot divider and other SRT-based radix-n dividers. The proposed USP-Awadhoot divider requires 266 slice logic LUTs and 146 slice register flip-flops. In contrast, the radix-2 to radix-16 divider implementations require 1500 to 2100 slice logic LUTs and 1100 to 1200 slice register flip-flops [7]. This indicates that the concept of different pre-scaling operations or factors for the input operands used in the proposed novel USP-Awadhoot divider helps to reduce its chip area requirements.

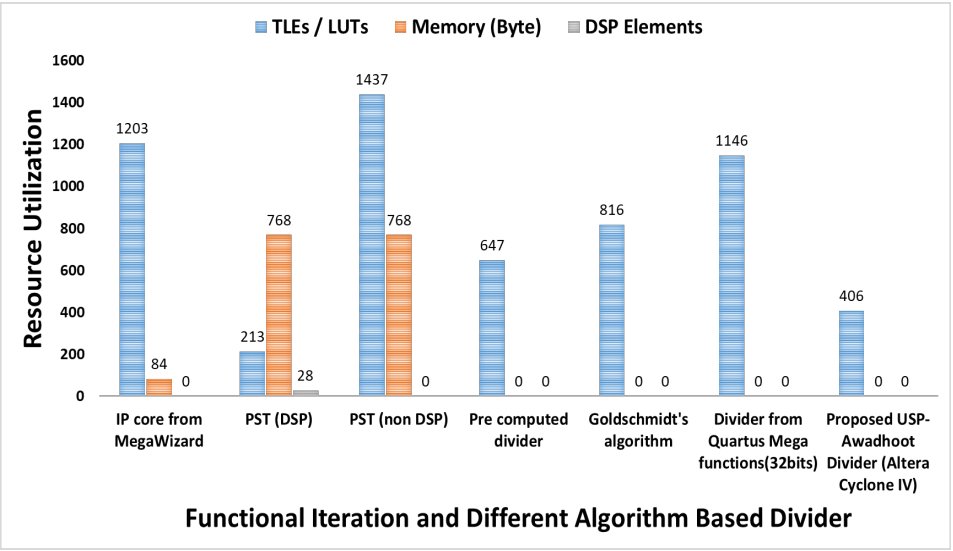


Figure 21. Comparative analysis of the proposed novel USP-Awadhoot divider with different functional dividers.

Figure 21 illustrates the comparative analysis of hardware resource utilization of the proposed USP-Awadhoot divider implementation with functional iteration and different algorithm-based dividers. For this comparison, we considered the implementation of the proposed USP-Awadhoot divider on Altera EP4CE6E22C8N Cyclone IV FPGA, as other dividers also implemented Altera FPGA and were feasible for other modern FPGA devices. The illustration considers the study by Md. F. Kasim, T. Adiono, Md. Fahreza and Md. F. Zakiy, which discussed a divider block with pre-computed values stored in a read-only memory as a look-up table [67]. J. Liu, M. Chang, and C-K. Cheng discussed the PST (DSP/non-DSP) algorithm that utilizes pre-scaling, series expansion, and Taylor series expansion together [33]. The comparison also considered the Mega Wizard IP core divider, Goldschmidt's algorithm-based divider, and the Quartus Mega function divider. The Altera EP4CE6E22C8N Cyclone IV FPGA implementation of the proposed novel USP-Awadhoot divider is used for the comparison. This implementation requires 406 LUTs / total logic elements (TLE) but zero memory and DSP elements.

In contrast, the PST (non-DSP) algorithm-based divider requires 213 TLEs / LUTs and 768 bytes of memory; whereas, the PST (DSP) algorithm-based divider needs 1437 TLEs / LUTs, 768 bytes of memory, and 28 DSP elements [33]. The pre-computed divider and Goldschmidt's algorithm-based divider require 647 and 816 TLEs / LUTs, respectively. The Mega Wizard IP core, DSP, and non-DSP dividers significantly delay the results, as their maximum clock frequencies are limited to 50-73 MHz. These dividers require relatively more resources than the proposed USP-Awadhoot divider. Additionally, the pre-computed values introduce rounding errors in the calculation process. The proposed USP-Awadhoot divider displays better implementation area requirements and maximum clock frequency performance.

Table 4 and Table 5 illustrate a resource consumption comparison of the proposed USP-Awadhoot divider and the Xilinx LogiCORE IP Divider Generator V4.0 [171]. Xilinx LogiCORE IP Divider Generator V4.0 creates a circuit for integer division based on a non-restoring radix-2 division algorithm or a high-radix division with pre-scaling [171].

The implementation statistics for the proposed USP-Awadhoot divider are obtained with Error and *Valid_O/P* signals. The *Valid_O/P* signal indicates that the computation has been completed, but the Error signal shows that an invalid condition has occurred due to a zero divisor value, i.e., the divide-by-zero condition. The proposed USP-Awadhoot divider implementation's resource requirements are analyzed with XilinxLogiCORE IP Divider Generator core V4.0, based on a non-restoring radix-2 division algorithm. Xilinx is the top candidate in the IC business, with a comprehensive set of Intellectual Property (IPs).

Table 4. Comparative analysis of the proposed USP-Awadhoot divider and the Xilinx LogiCORE IP integer divider generator V4.0 (8-bit).

Parameter / Result	Case 1	Case 2	Case 3	Case 4	Proposed USP-Awadhoot Divider
Dividend Width	8	8	8	8	8
Divisor Width	8	8	8	8	8
Remainder And Quotient Width	8	8	8	8	8
LUT6 -FF Pairs	223	218	217	215	000
No. of LUTs	203	205	203	197	266
No. of FFs	288	288	288	288	146
IC Name	Virtex7	Kintex7	Virtex6	Spartan6	Xilinx Zynq XC7Z010

Table 5. Comparative analysis of the proposed USP-Awadhoot divider and the Xilinx LogiCORE IP integer divider generator V4.0 (32-bit).

Parameter / Result	Case 1	Case 2	Case 3	Case 4	Proposed USP-Awadhoot Divider
Dividend Width	32	32	32	32	32
Divisor Width	32	32	32	32	32
Remainder And Quotient Width	32	32	32	32	32
LUT6 -FF Pairs	2196	2209	2195	2185	000
No. of LUTs	2068	2060	2126	2130	1836
No. of FFs	3202	3202	3202	3202	1352
IC Name	Virtex7	Kintex7	Virtex6	Spartan6	Xilinx Zynq XC7Z010

During the comparison, we considered Xilinx Virtex 6 and 7, Kintex 7, and Spartan 6 FPGA with a LogiCORE IP Divider Generator V4.0. The number of slice register flip-flops used in each FPGA IC is constant at 288, although the number of LUTs used changes from 197 to 205, and the number of six input LUT-FF pairs used changes significantly from 215 to 223, for 8-bit implementation in Virtex 7, Kintex 7, Virtex 6, and Spartan 6. Similarly, the number of slice register flip-flops used in each FPGA IC is constant at 3202, although the number of LUTs used changes from 2060 to 2130, and the number of six input LUT-FF pairs used changes significantly from 2185 to 2209, for 32-bit implementation in Virtex 7, Kintex 7, Virtex 6, and Spartan 6. The proposed USP-Awadhoot divider requires 266 to 1836 slice logic LUTs and 146 to 1352 slice register flip-flops. The power consumption of the LogiCORE IP Divider Generator V4.0 is not mentioned in the document, but the recommended divider based on the USP-Awadhoot division algorithm simulation estimates 3.366 Watts.

Based on the statistics presented in [9], the proposed novel USP-Awadhoot divider implementation shows improvements in its FPGA resource utilization, i.e., 77-88% improvement in the number of required slice logic LUTs (depending on the use of 8-bit

or 16-bit operands) and 96-96.36% improvement in the number of slice register flip-flops required (depending on the use of 8-bit or 16-bit operands) in the Xilinx IP core pipelined divider. The proposed novel USP-Awadhoot divider implementation uses 266 to 622 slice logic LUTs and 146 to 241 slice register flip-flops, depending on 8-bit or 16-bit operands, compared to the 2247 to 2742 slice logic LUTs and 4020 to 4904 slice register flip-flops of the IP core pipelined divider by Xilinx.

In [6], K. Tatas, D. J. Soudris, D. Siomos, M. Dasygenis, and A. Thanailakis discussed different concepts involved in partitioning the actual dividend into segments, to represent an actual division of a numerator by a denominator as a series of smaller divisions with a necessary requirement for the numerator to meet (Numerator *N*= N_1 + N_2 +...), as in equations (28) and (29). All intermediate operations are performed by considering the weight of the dividend bits. This concept of a series of divisions, showcases a smaller dividend division algorithm, where we must perform shifting, partisan division, and accumulation operations. Any existing division algorithm can be utilized for the small division process; the best-suited option must be selected depending on the trade-off between cost and area. This algorithm can be implemented in both series and parallel ways [6]. At last, add all the small division's results to get a combined result.

$$\frac{N}{D} = \frac{N_1}{D} + \frac{N_2}{D} + \frac{N_3}{D} + \frac{N_4}{D} + \dots\dots\dots \tag{28}$$

$$where, \qquad N = N_1 + N_2 + N_3 + \cdots \tag{29}$$

This algorithm is implemented with an N=32-bit dividend and parallel array divider, a sequential divider with two partitions, or a parallel divider with two sections in the partial division stage. Its respective implementation requires 4316, 2136, and 3050 slices on Xilinx Virtex-E 1000. From the above data, it is clear that the sequential implementation of this algorithm requires more hardware resources. Compared to this divider, the proposed USP-Awadhoot divider does not partition the given Numerator (Dividend) into smaller dividends, like N_1 and N_2, which requires the following $N = N_1 + N_2$ relation. In contrast, the proposed USP-Awadhoot divider partitions the numerator (dividend) into group dividends that do not need to sum to the actual dividend value, as shown in the equations. (30) and (31).

$$\begin{aligned} &Thus, Numerator = Dividend = X \\ &= group\ dividend\ 1, group\ dividend\ 2, \dots\dots, group\ dividend\ n \end{aligned} \tag{30}$$

$$\begin{aligned} Where, \qquad X \neq\ & group\ dividend\ 1 + group\ dividend\ 2 + \cdots \\ & + group\ dividend\ n \end{aligned} \tag{31}$$

Table 6. Summary of the comparison between standalone divider implementations year – 2019 [5].

Standalone Divider Implementation	Cycles		LUTs	FFs	Clock Frequency (MHz)
	Min.	Max.			
Radix-8 (8-bit) (Xilinx Virtex UltraScale)	11	11	500	75	475
Radix-16 (16-bit) (Xilinx Virtex UltraScale)	8	8	700	200	320
Quick-Div Initial (Xilinx Virtex UltraScale)	1	32	300	100	300
Quick-Div count leading zeros (Xilinx Virtex UltraScale)	2	33	350	170	400
Quick-Div CLZ-2BIT worst-case optimization (Xilinx Virtex UltraScale)	2	33	450	170	300
Proposed USP-Awadhoot divider—8 bits (Xilinx Zynq XC7Z010)	---	---	266	146	285
Proposed USP-Awadhoot divider—16 bits (Xilinx Zynq XC7Z010) version V16.1	---	---	622	241	125

In [5], E. Matthews, A. Lu, Z. Fang, and L. Shannon discussed integer divider designs for FPGA-based soft processors that influence the use of variable latency execution units in their instruction pipeline. Implementation efforts focused on the Quick-Div divider, which exhibits data dependency and variable latency in integer division. It is optimized on FPGA and integrated into the Taiga RISC-V pipelined soft processor. They pointed out that a 64-bit floating/fixed-point divider requires almost ten times more resources than a 32-bit radix-2 integer divider [5, 168]. FPGA soft-core processors such as Micro Blaze [152], NIOS II [169], and the LEON3 processor [170], implemented fixed-latency radix-2 dividers with 32 cycles of latency for performing division operations. As illustrated in Table 6, experimental implementations of the Quick-Div divider are performed over the Xilinx Virtex UltraScale + VCU118 board (XCVU9P-L2FLGA2104E) using the Vivado 2018.3 synthesis tool. It also provides comparative statistics between the data-dependent variable-latency Quick-Div dividers stand-alone implementation and fixed-latency radix-n (n = 8, 16) divider implementations. Here, frequency is the divider clock frequency used for the division operation.

Table 7. Summary of the Taiga soft processor divider implementation comparison year – 2019 [5].

Taiga Soft Processor Divider Implementation	LUTs	FFs	Clock Frequency (MHz)
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
Proposed USP-Awadhoot divider—8 bits	266	146	285
Proposed USP-Awadhoot divider—16 bits	622	241	125

All dividers are realized with the RISC-V Taiga soft processor on Xilinx FPGA by influencing variable latency execution units in their instruction pipeline. As illustrated in Table 7, a comparative statistic is derived between the implementation of data-dependent variable-latency Quick-Div dividers with the Taiga RISC-V soft-processor and fixed latency radix-n (n = 8, 16) dividers. Considering Table 6 and Table 7, the variable-latency Quick-Div dividers and fixed-latency radix-n (n = 8, 16) dividers require 5 to 7 times more chip area than the proposed USP-Awadhoot divider, which is represented by the number of LUTs and flip-flops used for implementation. In contrast, the maximum clock frequency of variable-latency Quick-Div dividers and fixed-latency radix-n (n = 8, 16) dividers is almost double that of the maximum clock frequency of the proposed novel USP-Awadhoot divider. This indicates that the proposed novel USP-Awadhoot divider implementation achieved improvements in FPGA resource utilization.

To summarize, compared to the existing state-of-the-art digit recurrence dividers, the proposed novel USP-Awadhoot divider implements multiple performance improvement techniques simultaneously (i.e., dynamic separate scaling operations) with individual input operands. This approach achieves variable latency and a small area footprint without overlapping regions in the quotient calculation logic. To implement and test the proposed novel USP-Awadhoot divider, I used the Vivado simulation tool with the Zynq XC7Z010 and XCZU7EV-FFVC1156-2-E Xilinx FPGAs, as well as the Quartus Prime Lite simulation software with the EP4CE6E22C8N Cyclone IV Altera FPGAs. All comparisons were conducted using the same synthesis tools and FPGA platforms. More specifically, if another divider was implemented using the Xilinx Vivado synthesis tool with Xilinx FPGAs, the proposed novel USP-Awadhoot divider implementation was compared on the same platform. Similarly, if another divider was implemented using the Altera Quartus synthesis tool with Altera FPGAs, comparisons were conducted using the same Altera Quartus synthesis tool with the Altera FPGA platform.

5.4 Chapter conclusion

A detailed explanation of the proposed novel USP-Awadhoot divider implementation statistics, verification test strategies, waveform analysis, and comparative analysis is described in this chapter. This demonstrates how the proposed USP-Awadhoot divider improves divider implementation, especially regarding implementation area requirements. This chapter covers RO4 – “Implement the divider based on the formulated algorithm and improve the area requirements to compose the operand-dependent area-efficient divider circuit design”. Publications I, II, and IV cover detailed information on the proposed novel USP-Awadhoot divider implementation statistics, verification test strategies, waveform analysis, and comparative analysis.

Conclusion and future work prospects

The evaluation of addition and multiplication implementations typically falls within a latency range of a few clock cycles to less than ten, while the performance evaluation of division operations usually spans tens to hundreds of clock cycles and requires a significantly larger implementation area. The main objective of this doctoral dissertation is to address the research gap in simultaneously applying multiple performance-enhancement techniques to individual input operands, with the goal of designing and implementing a divider circuit that minimizes implementation area, avoids rounding errors, and prevents overlapping regions in the quotient bit calculation logic. The details presented in Chapters 3 and 5 confirm the successful implementation of the proposed novel USP-Awadhoot divider circuit block. The USP-Awadhoot divider integrates multiple performance-enhancing techniques (dynamic separate scaling operations) for each input operand, achieving variable latency and preventing overlapping regions in the quotient bit calculation logic. The design is developed by simulating the proposed technique and cross-verifying the results against standard truth tables, which include all possible combinations of input operands, generated from a theoretical evaluation of the proposed idea. The main contributions highlighted in the thesis are as follows:

- In association with RO1, RO2, RO3, and RO4, this thesis contributes to the development of a novel algorithm for implementing a divider circuit block. The innovative concept of dynamic separate scaling operations for the dividend and divisor reduces resource requirements, resulting in a divider circuit block with a low area footprint.
- In association with RO2, RO3, and RO4, I developed an easy Group Quotient (GQ_n) value selection logic in the proposed divider circuit block based on the unique relation derived between Dividend Groups (GD_d), Modified Divisor (MD_r), and Flag Digit (*FD*) without any critical overlapping.
- In association with RO2, RO3, and RO4, I developed a clear process for selecting the final quotient based on the Group Quotient (GQ_n), Partial Quotient (PQ_n), and Additional Quotient (*AQ*) values without critical overlapping regions.
- In association with RO2 and RO3, I implemented a complex divider based on the Baudhayan-Pythagorean triplet algorithm with the proposed USP-Awadhoot divider circuit block.
- The described steps reduce the criticality of the conversion logic by eliminating overlapping regions in the quotient bit selection logic.

Future work prospects

- As the current implementation verifies the successful performance of the proposed divider on different FPGAs, the next target is to design a dedicated chip. The first step is to develop a physical layout and floor plan for the chip-level implementation, which determines where each circuit component is placed and extracts parasitic values to prepare the final design for fabrication.

- To develop a speed-oriented USP-Awadhoot algorithm-based divider circuit that achieves low latency and conversion time, we need to fuse certain intermediate functional blocks. For example, separate operations like addition and multiplication can be performed in a fused mode, such as fused multiply-add (FMA). Additionally,

the focus should be on improving the 24-bit and 32-bit implementations of the proposed divider.

- The current implementation validated its successful performance by using combinational circuits. Certain processes in the proposed divider can be represented using different hardware architectures, such as pipelined architecture, parallel architecture, array structure, and cascade structure.
- The performance of the proposed divider implementation needs to be verified in various applications such as image processing, particle detection, telecommunication, and signal processing.

References

[1] Bailey, D. G. "Space-Efficient Division on FPGAs," Electronics New Zealand Conference, p.p.- 206-211, 2006.

[2] Qasaimeh, M., Denolfy. K., Loy, J. Vissersy, K. Zambreno, J. and Jones, P. H. Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels. Proc. Int. Conf. on Embedded Software and Systems. (ICESS) 1–8, 1906.11879v1, 2019.

[3] Kumari, J. and Yasin, M. Y. Design and soft implementation of n-bit SRT divider on FPGA through VHDL. Int. Jou. for Innovations in Eng. Sci. and Management. Vol. 3, issue 4, 13-19, April 2015.

[4] Narendra, K., Ahmed, B. S., Kumar, K. S. and Asha, G. H. FPGA Implementation of fixed-point integer divider using iterative array structure. Int. Jou. of Eng. and Tech. Research – IJETR. Vol. 3, issue 4, 170-179, 2321-0869, April 2015.

[5] Matthews, E., Lu, A., Fang, Z. and Shannon, L. Rethinking integer divider design for FPGA-based soft-processors. IEEE 27th Annual Int. Symp. on Field-Programmable Custom Comp. FCCM. 289-297, 10.1109/FCCM.2019.00046, 2019.

[6] Tatas, K., Soudris, D. J., Siomos, D., Dasygenis, M. and Thanailakis, A. A Novel division algorithm for parallel and sequential processing. 9th Int. Conf. on Elecs., Cir., and Sys. 553-556, 0-7803-7596-3, 2002.

[7] Tocher, K. D. Techniques of multiplication and division for automatic binary computers. Quart. Journ. Mech. and Applied Math. Vol. XI, pt. 3, 364-384, 1958.

[8] Asai, H. A recursive radix conversion formula and its application to multiplication and division. Comp. and Maths. With Applications.Vol. 2, 255-265, 1976.

[9] Sorokin, N. Implementation of high-speed fixed-point dividers on FPGA. Jou. of Com. Sci. & Tech. Vol. 6, no. 1, 8-11, 1666-6038, April 2006.

[10] Huang, K. and Chen, Y. Improving performance of floating-point division on GPU and MIC. Proc. of 15th Int. Con. on Algo. and Arch. for Parallel Proc.- ICA3PP. 691-703, 10.1007/978-3-319-27122-4 48, Nov.2015.

[11] Fang, X. and Leeser, M. Vendor agnostic, high performance, double-precision floating-point division for FPGAs. Proc. of IEEE High-Perf. Extreme Com. Conf., HPEC, pp. 1-5. 10.1109/HPEC.2013.6670335, Sep.2013.

[12] Liu, W. and Nannarelli, A. Power dissipation challenges in multicore floating-point units. Proc. of 21st IEEE Int. Con. on App.-spec. Sys. Archit. and Proc. 257-264, 10.1109/ASAP.2010.5540986, 2010.

[13] Thall, A. Extended-precision floating-point numbers for GPU computation. Proc. of Spec. Interest Grp. on Comp. Graphics and Interactive Tech. Con.- SIGGRAPH06. 978-1-59593-364-5, 2006.

[14] Sinha, P. Smart Sensors use DSCs for Embedded Signal Processing. Jour. Microchip Technology Inc. 2021. https:// cdn. Weka-fachmedien.de/whitepaper/files/ 108_mca471wp_sensor signal processing with dscs.pdf.

[15] Oberman, S. F., and Flynn, M. J. Division algorithms and implementations. IEEE Trans. on Computers, Vol. 46, no. 8, 833-854, August - 1997.

[16] Pin˜eiro, J. A., Bruguera, J. D., Lamberti, F. and Montuschi, P. A radix-2 digit-by-digit architecture for cube root. IEEE Trans. On Computers, Vol. 57, No. 4, 562-566, April - 2008.

[17] Takagi, N., Kadowaki, S. and Takagi, K. A hardware algorithm for integer division. Proc. of the 17th IEEE Symp. on Com. Arit.-ARITH. 1-7, pp. 140-146, 1063-6889/05 (2005).

[18] Lee, B.R. and Burgess, N. Improved small multiplier based multiplication, squaring, and division. Proc. of the 11th Annual IEEE Symp. on Field-Prog.Custom Com.- FCCM, pp. 91–97, 1082-3409/03, 2003.

[19] Nannarelli, A. and Lang, T. Low-power divider. IEEE Tran. On Com. Vol. 48, No. 1, pp. 2-14, 0018-9340/99, 1999.

[20] G. Sutter, G. Bioul, J-P Deschamps, "Comparative Study of SRT-Dividers in FPGA," Becker J., Platzner M., Vernalde S. (eds) Field Programmable Logic and Application. FPL 2004. Lecture Notes in Computer Science, vol 3203. Springer, Berlin, Heidelberg, p.p – 209-220.

[21] M. Reddy, Vasantha MH, N. Kumar, D. Dwivedi, "Design of Approximate Dividers for Error Tolerant Applications" IEEE 61st International Midwest Symposium on Circuits and Systems- MWSCAS ISBN- 978-1-5386-7392-8/18, 2018, p.p- 496 – 499.

[22] J-S. Chiang, H-Da. Chung and M-S. Tsai, "Carry-Free Radix-2 Subtractive Division Algorithm and Implementation of the Divider," Tamkang Journal of Science and Engineering, Vol. 3, No. 4, 2000, pp. 249-255.

[23] 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, Technical Report CSL-TR-93-554, January 1993.

[24] N. Burgess, "A Fast Division Algorithm for VLSI," IEEE International Conference on Computer Design: VLSI in Computers and Processors, Cambridge, MA, USA, 14-16 Oct. 1991, p.p- 560-563.

[25] M. Kuhlmann, and K. K. Parhi, "Fast Low-Power Shared Division and Square-Root Architecture," Proceedings International Conference on Computer Design. VLSI in Computers and Processors, 1998, p.p.- 128-135.

[26] 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. Research and development, vol. 11, pp. 34-53, Jan. 1967, doi: 10.1147/rd.111.0034.

[27] D. L. Fowler and J. E. Smith, "An Accurate, High-Speed Implementation of Division by Reciprocal Approximation," Proc. Ninth IEEE Symp. Computer Arithmetic, pp. 60-67, Sept. 1989, doi: 10.1109/ARITH.1989.72810.

[28] B. Liebig, and A. Koch, "Low-Latency Double-Precision Floating-Point Division for FPGAs," International Conference on Field-Programmable Technology (FPT), Shanghai, China, 10-12 December 2014, p.p-107-114.

[29] K. N. Han, A. F. Tenca, and D. Tran, "High-speed floating-point divider with the reduced area," Proc. SPIE 7444, Mathematics for Signal and Information Processing, CCC code: 0277-786X/09, doi: 10.1117/12.827850, 2009, p.p- O.1-O.8.

[30] I. Kong, and E. E. Swartzlander, Jr., "A Goldschmidt Division Method With Faster Than Quadratic Convergence," IEEE Transactions On Very Large Scale Integration (Vlsi) Systems, Vol. 19, No. 4, April 2011, p.p- 696-700.

[31] R. E. Goldschmidt, "Applications Of Division By Convergence," Masters Degree Thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, June 1964, URI: http://hdl.handle.net/1721.1/11113.

[32] T. J. Kwon, J. Sondeen, and J. Draper, "Floating-Point Division and Square Root using a Taylor-Series Expansion Algorithm," 50th Midwest Symposium on Circuits and Systems, Montreal, Que., Canada 5-8 Aug. 2007, DOI-10.1109/MWSCAS.2007.4488594, p.p- 305-308.

[33] J. Liu, M. Chang, and C.-K. Cheng, "An Iterative Division Algorithm for FPGAs," 14th Int. Symp. on Field programmable gate arrays FPGA'06, Feb. 22–24, 2006, Monterey, California, USA, ACM 1595932925/06/0002, pp. 83-89.

[34] A. Kumar, and T. N. Sasamal, "Design of Divider Using Taylor Series in QCA," 1st International Conference on Power Engineering, Computing and Control, PECCON-2017, Chennai, India, Energy Procedia 117, 818-825, p.p.- 818-825

[35] A. A. Liddicoat and M. J. Flynn, "High-Performance Floating-Point Divide," Proceedings Euromicro Symposium on Digital Systems Design, Warsaw, Poland, 4-6 Sept. 2001, p.p- 354-361.

[36] Trummer, R., Zinterhof, P. and Trobec, R. A high-performance data-dependent hardware divider. Chapter 7: Systems and Simulation, Parallel Numerics, 961-6303-67-8, 193-206, Uni. of Salzburg 2005.

[37] R. K. L. Trummer, "A High-Performance Data-Dependent Hardware Integer Divider," Master thesis, Institute of Computer Science and Systems Analysis, Paris Lodron University, Salzburg, May 2005.

[38] Britannica, The Editors of Encyclopaedia. "Hindu-Arabic numerals." Encyclopedia Britannica, 8 September 2017, https://www.britannica.com/topic/Hindu-Arabic-numerals.

[39] The Arabic Numeral System (MacTutor by the School of Math. and Stat. at the University of St Andrews, Scotland). Available online-https://mathshistory.st-andrews.ac.uk/HistTopics/Arabic_numerals/.

[40] Brahmagupta (MacTutor by the School of Math. and Stat. at the University of St Andrews, Scotland). Available online- https://mathshistory.st-andrews.ac.uk/Biographies/Brahmagupta.

[41] Robertson, E. and O'Connor, J. Aryabhata, the Elder. MacTutor by the School of Math. and Stat. at the University of St Andrews, Scotland. https://mathshistory.st-andrews.ac.uk/Biographies/Aryabhata_I.

[42] A. Kaplan, Math on Call: A Mathematics Handbook. Wilmington, MA, USA: Great Source Education Group, 2004.

[43] T. Bassarear and M. Moss, Mathematics for Elementary School Teachers, 4th ed. Independence, KY, USA: Cengage Learning, 2008.

[44] Detrey, J., and Dinechin, F. D. A tool for unbiased comparison between logarithmic and floating-point arithmetic. Jou. of VLSI Signal Processing, 49, 161–175, 10.1007/s11265-007-0048-7, 2007.

[45] 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.

[46] M. D. Ercegovac and J-M Muller "Design of a complex divider," Proc. SPIE 5559, Advanced Signal Processing Algorithms, Architectures, and Implementations XIV, 26 October 2004; p.p- 51-59, https://doi.org/10.1117/12.560154D.

[47] M. M. Kermani, N. Manoharan, and R. Azarderakhsh, "Reliable Radix-4 Complex Division For Fault Sensitive Applications," IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, Vol.- 34, No.- 4, April 2015, p.p- 656-667.

[48] D. Wang, P. Ren, and L. Liu, "A High- Throughput Fixed-Point Complex Divider For FPGA," IEICE Electronics Express Letter, Vol. 10, No.4, p.p- 1-8, DOI: 10.1587/elex.10.20120879.

[49] 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.

[50] 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.

[51] B. K. Bose, L. Pei, G. S. Taylor, and D. A. Patterson, "Fast multiply and divide for a VLSI floating-point unit," 1987 IEEE 8th Symposium on Computer Arithmetic (ARITH), 1987, pp. 87-94, doi: 10.1109/ARITH.1987.6158684.

[52] H. Nikmehr, "Architectures for Floating-Point Division," Ph.D. thesis, School of Electrical and Electronic Engineering, The University of Adelaide, Australia, August 2005.

[53] P. Soderquist, and M. Leeser, "Division and square root: choosing the right implementation," IEEE Micro, Vol-17, p.p- 56-66, 1997.

[54] E. N. Frantzeskakis and K. J. R. Liu, "A class of square root and division free algorithms and architectures for QRD-based adaptive signal processing," in IEEE Transactions on Signal Processing, vol. 42, no. 9, pp. 2455-2469, Sept. 1994, doi: 10.1109/78.317867.

[55] D. Piso, J. A. Pineiro, and J. D. Bruguera, "Analysis of the Impact of Different Methods for Division/Square root Computation in the Performance of a Superscalar Microprocessor," Journal of Systems Architecture, 49 (12-15): 543-555, December 2003.

[56] Reza Z, Mehdi K, Arash F, Ali Kusha, Saeed S, Massoud P, "SEERAD: A High Speed yet Energy-Efficient Rounding based Approximate Divider" Design, Automation & Test in Europe Conference & Exhibition – DATE 2016, ISSN: 978-3-9815370-7-9, p.p- 1481-1484.

[57] Elizabeth A, Suganthi V, and Seok-Bum Ko "Approximate Restoring Dividers Using Inexact Cells and Estimation From Partial Remainders" Ieee Transactions On Computers, Vol. 69, No. 4, April 2020, p.p- 468-474.

[58] L. Chen, J. Han, W. Liu, F. Lombardi, "Design of Approximate Unsigned Integer Non-restoring Divider for Inexact Computing" Great Lakes Symposium on VLSI, GLSVLSI- 2015, Pittsburgh Pennsylvania USA, ISBN- 978-1-4503-3474-7, May 2015, p.p- 51-56.

[59] N. Jamadagani, Jo Ebergen, "An Asynchronous Divider Implementation" IEEE 18th International Symposium on Asynchronous Circuits and Systems, 1522-8681/12, 2012 IEEE, p.p- 97-104.

[60] P. Saha, D. Kumar, P. Bhattacharyya, A. Dandapat, "Vedic division methodology for high-speed very-large-scale integration applications" Journal of Engineering; Accepted on 7 January 2014 Vol. 2014, Iss. 2, pp. 51–59.

[61] M. Reddy, Vasantha MH, N. Kumar, D. Dwivedi, "Design of Approximate Dividers for Error Tolerant Applications" IEEE 61st International Midwest Symposium on Circuits and Systems- MWSCAS ISBN- 978-1-5386-7392-8/18, 2018, p.p- 496 – 499.

[62] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms," third edition, The MIT Press, Cambridge, Massachusetts London, England. ISBN 978-0-262-03384-8 (hardcover)—ISBN 978-0-262-53305-8 (pdf).

[63] O. L. MacSorley, "High-Speed Arithmetic in Binary Computers," Proceedings of the IRE, 49, p.p- 67-90, January 1961.

[64] G. Metze, "A class of binary divisions yielding minimally represented quotients," IRE Trans. Electronic Computers, vol. EC-11, pp. 761-764, December 1962.

[65] P. Soderquist and M. Leeser, "Area and performance tradeoffs in floating-point divide and square-root implementations," ACM Comput. Surveys, Vol - 28, Issue – 3, p.p- 518–564, September 1996. DOI:https://doi.org/10.1145/243439.243481.

[66] S. Dixit and M. Nadeem, "FPGA accomplishment of a 16-bit divider," Imperial J. Interdiscipl. Res., vol. 3, no. 2, pp. 140–143, 2017.

[67] M. F. Kasim, T. Adiono, M. Fahreza, and M. F. Zakiy, "FPGA implementation of fixed-point divider using pre-computed values. 4th Int. Conf. on Electrical Eng. and Informatics ICEEI. Vol. 11, 206-211, 2013.

[68] 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.

[69] 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.

[70] 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.

[71] Behrooz Parhami, "Computer Arithmetic: Algorithms and Hardware Designs," book, Oxford University Press, 2010 - 641 pages.

[72] Miloš D. Ercegovac and Tomás Lang, "Digital Arithmetic A volume in The Morgan Kaufmann Series in Computer Architecture and Design," Imprint: Morgan Kaufmann, 2004, ISBN- 978-1-55860-798-9.

[73] K. Jun, "Modified non-restoring division algorithm with improved delay profile," M.S. thesis, Fac. Graduate, School Univ. Texas Austin, Austin, TX, USA, 2011.

[74] J. Cocke and D. W. Sweeney, “High-Speed Arithmetic in a Parallel Device,” Technical Report, IBM Corp., February 1957.

[75] J. E. Robertson, ‘‘A new class of digital division methods,’’ IRE Trans. Electron. Comput., vol. EC- 7, issue no. 3, pp. 218–222, Sep. 1958.

[76] M. Nadler, “A High-Speed Electronic Arithmetic Unit for Automatic Computing Machines,” Alta Technica (Prague), Vol- 6, p.p.- 464-478, 1956.

[77] O. L. MacSorley, “High-Speed Arithmetic in Binary Computers,” Proceedings of The IRE, Vol- 49, p.p- 67-91, January 1961.

[78] J. B. Wilson and R. S. Ledley, “An Algorithm for Rapid Binary Division,” IRE Transactions on Electronic Computers, EC-10, p.p- 662-670, 1961.

[79] G. Metze, “A Class of Binary Divisions,” IRE Transactions on Electronic Computers, EC-11, Vol-6, 1962.

[80] L. Ciminiera and P. Montuschi, “Simple Radix 2 Division and Square Root with Skipping Some Addition Steps,” Proceedings of The 10th IEEE Symposium on Computer Arithmetic (ARITH-10’91), p.p.-202-209, Grenoble, France, 26-28 June 1991, IEEE Computer Society Press.

[81] D. M. Mandelbaum, “A systematic Method for Division with High Average Bit Skipping,” IEEE Transactions on Computers, Vol.- 39, Issue- 1, p.p.- 127-130, 1990.

[82] D. E. Atkins, “The Theory and Implementation of SRT Division,” Technical Report UIUCDCS-R-67-230, Department of Computer Science, the University of Illinois at Urbana-Champaign, Urbana, Illinois, USA, 1967.

[83] 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.

[84] N. Takagi and S. Kuwahara, “Digit-recurrence algorithm for computing Euclidean norm of a 3-D vector,” ARITH '99: Proceedings of the 14th IEEE Symposium on Computer Arithmetic, April 1999. DOI: 10.1109/ARITH.1999.762833.

[85] D. L. Harris, S. F. Oberman, and M. A. Horowitz, "SRT Division: Architectures, and Implementations" Proceedings 13th IEEE Symposium on Computer Arithmetic, Asilomar, CA, USA, July 6-9, 1997, 0-8186-7846-1. DOI.10.1109/ARITH.1997.614875.

[86] Sumiksha, P. Konda, and S. Shetty, "Computation of SRT and CORDIC Division Algorithms," IOSR Journal of Electronics and Communication Engineering (IOSR-JECE), vol. 12, issue 4, ver. II, pp. 53-56, July-Aug. 2017, e-ISSN: 2278-2834, ISSN: 2278-8735.

[87] D. Tyanev and Y. Petkova, “Hardware Divider,” CompSysTech'18: Proceedings of the 19th International Conference on Computer Systems and Technologies, P.p.-139–143, September 2018, https://doi.org/10.1145/3274005.3274009.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] R. E. Bryant, ‘‘Bit-level analysis of an SRT divider circuit,’’ in Proc. 33rd Design Automation Conf., Las Vegas, NV, USA, 1996, pp. 661–665.

[93] 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.

[94] T. M. Carter and J. E. Robertson, ‘‘Radix-16 signed-digit division,’’ IEEE Trans. Comput., vol. 39, no. 12, pp. 1424–1433, Dec. 1990.

[95] 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.

[96] 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.

[97] E.M. Clarke, S.M. German, and X. Zhao, “Verifying the SRT Division Algorithm Using Theorem Proving Techniques,” In Alur R., Henzinger T.A. (eds) Computer Aided Verification. CAV 1996. Lecture Notes in Computer Science, vol 1102. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-61474-5_62, p.p- 111-122.

[98] 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.

[99] S. Knowles, ‘‘Arithmetic processor design for the T9000 transputer,’’ Proc. SPIE, vol. 1566, pp. 230–243, Dec. 1991.

[100] A. Nannarelli, ‘‘Performance/power space exploration for binary64 division units,’’ IEEE Trans. Comput., vol. 65, no. 5, pp. 1671–1677, May 2016.

[101] 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.

[102] M. D. Ercegovac and T. Lang, ‘‘Simple radix-4 division with operands scaling,’’ IEEE Trans. Comput., vol. 39, no. 9, pp. 1204–1208, Sep. 1990.

[103] W. Liu and A. Nannarelli, ‘‘Power efficient division and square root unit,’’ IEEE Trans. Comput., vol. 61, no. 8, pp. 1059–1070, Aug. 2012.

[104] 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.

[105] 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.

[106] 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.

[107] 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.

[108] S. E. McQuillan, J. V. McCanny, and R. Hamill, ‘‘New algorithms and VLSI architectures for SRT division and square root,’’ Proc. IEEE 11th Symposium Computer Arithmetic, Jul. 1993, pp. 80–86.

[109] 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.

[110] 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.

[111] 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.

[112] 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.

[113] G. S. Taylor, ‘‘Radix 16 SRT dividers with overlapped quotient selection stages,’’ in Proc. 7th IEEE Symp. Comput. Arithmetic, Jun. 1985, pp. 64–71.

[114] T. E. Williams and M. A. Horowitz, ‘‘A zero-overhead self-timed 160- ns 54-b CMOS divider,’’ IEEE J. Solid-State Circuits, vol. 26, no. 11, pp. 1651–1661, Nov. 1991.

[115] T. Asprey, G.S. Averill, E. Delano, R. Mason, B. Weiner, and J. Yetter, “Performance Features of the PA7100 Microprocessor,” IEEE Micro, vol. 13, no. 3, pp. 22-35, June 1993.

[116] T. Lynch, S. McIntyre, K. Tseng, S. Shaw, and T. Hurson, “High-Speed Divider with Square Root Capability,” U.S. Patent No. 5,128,891, 1992.

[117] D. Hunt, “Advanced Performance Features of the 64-bit PA-8000,” Digest of Papers COMPCON ’95, pp. 123-128, Mar. 1995.

[118] A. Svoboda, “An Algorithm for Division,” Information Processing Machines, vol. 9, pp. 29-34, 1963.

[119] C. Tung, ‘‘A division algorithm for signed-digit arithmetic,’’ IEEE Trans. Comput., vol. C-17, no. 9, pp. 887–889, Sep. 1968.

[120] L. A. Montalvo, K. K. Parhi, and A. Guyot, ‘‘New Svoboda-Tung divi- sion,’’ IEEE Trans. Comput., vol. 47, no. 9, pp. 1014–1020, Sep. 1998.

[121] J. S. Chiang and M.-S. Tsai, ‘‘A radix-4 new Svobota-Tung divider with constant timing complexity for prescaling,’’ J. VLSI Signal Process., vol. 33, pp. 117–124, Jan. 2003.

[122] L. Montalvo and A. Guyo, ‘‘Svoboda-Tung division with no compensation,’’ in Proc. IEEE Int. Conf. VLSI Design, Jan. 1995, pp. 381–385.

[123] 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.

[124] 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.

[125] 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.

[126] 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.

[127] 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.

[128] 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.

[129] 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.

[130] 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.

[131] 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.

[132] M. D. Ercegovac and R. McIlhenny, ‘‘Design and FPGA implementation of a 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.

[133] 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.

[134] 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.

[135] 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.

[136] 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.

[137] J. D. Bruguera, ‘‘Radix-64 floating-point divider,’’ in Proc. IEEE 25th Symp. Comput. Arithmetic (ARITH), Jun. 2018, pp. 84–91.

[138] D. Das Sarma and D. W. Matula, ‘‘Faithful bipartite ROM reciprocal tables,’’ in Proc. 12th Symp. Comput. Arithmetic, Jul. 1995, pp. 12–25.

[139] 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.

[140] 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.

[141] 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.

[142] 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.

[143] 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.

[144] 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.

[145] 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.

[146] T. Williams, N. Parkar, and G. Shen, “SPARC64: A 64-b 64-Active-Instruction Out-of-Order-Execution MCM Processor,” IEEE J. Solid-State Circuits, vol. 30, no. 11, pp. 1,215-1,226, Nov. 1995.

[147] S. E. Richardson, ‘‘Exploiting trivial and redundant computation,’’ Proceedings of IEEE 11th Symp. Computer Arithmetic, July. 1993, pp. 220–227.

[148] 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.

[149] 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.

[150] 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.

[151] T. Jebelean, ‘‘An algorithm for exact division,’’ J. Symbolic Comput., vol. 15, no. 2, pp. 169–180, Feb. 1993.

[152] MicroBlaze Processor Reference Guide, Xilinx Inc. [Online]. Available: xilinx.com/ support/ documentation /sw manuals/xilinx2016 4/ug984- vivado-MicroBlaze-ref.pdf.

[153] S. M. Mueller, and S. D. Trong, “Optimized Structure For Hexadecimal and Binary Multiplier Array,” United State Patent, Patent No.: US 9.720.648 B2, August 2017.

[154] J. C. Ebergen, N. P. Jamadagni, and I. E. Sutherland, “Performing Quotient Selection for A Carry-Save Division Operation,” United States Patent, Patent No.: US 9.298.421 B2, March 2016.

[155] P. Kubinec, J. Púčik, M. Hagara, E. Cocherová, and O. Ondráček, "Successive Approximation Algorithm for Complex Number Magnitude and Argument Computation,"

30th International Conference Radioelektronika-2020, pp.1-4, doi: 10.1109 /radioelektronika 49387.2020.9092432.

[156] T. Aoki, H. Amada and T. Higuchi, "Real/complex reconfigurable arithmetic using redundant complex number systems," Proceedings 13th IEEE Symposium on Computer Arithmetic, 1997, pp. 200-207, doi: 10.1109/ARITH.1997.614896.

[157] M. M. A. Basiri and N. M. Sk, "An efficient hardware-based MAC design in digital filters with complex numbers," 2014 International Conference on Signal Processing and Integrated Networks (SPIN), 2014, pp. 475-480, doi: 10.1109/SPIN.2014.6777000.

[158] T. Jamil, "An Introduction to Complex Binary Number System," 2011 Fourth International Conference on Information and Computing, 2011, pp. 229-232, doi: 10.1109/ICIC.2011.37.

[159] Zaini, H. and R. G. Deshmukh. “A novel method for arithmetic operations using complex binary number system and the reconversion of the result to the decimal complex number system.” IEEE SoutheastCon, 2003. Proceedings. (2003): 31-37.

[160] T. Jamil, "Complex Binary Associative Dataflow Processor - A Tutorial," SoutheastCon 2018, 2018, pp. 1-3, doi: 10.1109/SECON.2018.8478931.

[161] T. Aoki, Y. Ohki and T. Higuchi, "Redundant complex number arithmetic for high-speed signal processing," VLSI Signal Processing, VIII, 1995, pp. 523-532, doi: 10.1109/VLSISP.1995.527523.

[162] Y. Ohi, T. Aoki, and T. Higuchi, "Redundant complex number systems," Proceedings 25th International Symposium on Multiple-Valued Logic, 1995, pp. 14-19, doi: 10.1109/ISMVL.1995.513504.

[163] D. M. Priest, “Efficient scaling for complex division,” ACM Transactions on Mathematical Software, Volume-30, Issue-4, pp. 389–401, December 2004, https://doi.org/10.1145/1039813.1039814.

[164] Agrawala, Dr. V.S. (Ed.). & Jagadguru Swami Sri Bharati Krsna Tirthaji Maharaja, “Vedic Mathematics: Sixteen simple mathematical formulae from the Vedas,” Motilal Banarsidass: Delhi, 1988. ISBN-10: 8120801636, ISBN-13: 978-8120801639.

[165] J. J. O'Connor and E. F. Robertson, “The Indian Sulbasutras,” MacTutor by the School of Mathematics and Statistics, University of St Andrews, Scotland, https://mathshistory.st-andrews.ac.uk/HistTopics/Indian_sulbasutras/.

[166] U. S. Patankar, A. Koel, "Division Method and Circuit" PTC the International Patent System, International Bureau of the World Intellectual Property Organization, application no.: PCT/IB2021/054942, submission no.: 054942, Date: 06 June 2021; published on- 15-12-2022, publication no- WO2022259009. patentscope.wipo.int

[167] U. S. Patankar and S. M Patankar, "Elements of Vedic Mathematics" Book ISBN 9789949832163.

[168] 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.

[169] 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.

[170] GRLIB IP Core User’s Manual, Cobham Gaisler AB. Accessed: Aug. 2020. [Online]. Available: https://gaisler.com/products/grlib/grip. Pdf.

[171] Product Specification, "LogiCORE IP Divider Generator V4.0," Xilinx, Inc, DS819, June 2011, pg. 1-27.

[172] Xilinx user guide, "UltraScale Architecture Configurable Logic Block," Xilinx, Inc, UG574 (v1.5) February 28, 2017.

[173] AMD Xilinx technical reference manual, " Zynq UltraScale+ Device," AMD Xilinx, Inc, UG1085 (v2.3.1) January 4, 2023.

[174] G. Sutter and J. -P. Deschamps, "High speed fixed point dividers for FPGAs," 2009 International Conference on Field Programmable Logic and Applications, Prague, Czech Republic, 2009, pp. 448-452, doi: 10.1109/FPL.2009.5272492.

Acknowledgments

I would like to thank the Thomas Johann Seebeck Department of Electronics, Tallinn University of Technology, for providing me with this opportunity to pursue a Ph.D. degree as an individual researcher. I want to thank my supervisors, Dr. Ants Koel, Dr. Tamás Pardy, Prof. Toomas Rang, and Prof. Yannick Le Moullec, who guided me, assisted my journey, and motivated me throughout my Ph.D. studies. I am especially thankful to Dr. Ants Koel and Prof. Toomas Rang for supporting and nurturing my thoughts and ideas, and encouraging me to write my first book. They provided me with an exciting opportunity to work with the TTU satellite design team and encouraged my participation in different semiconductor manufacturing training, such as the Rochester Institute IC design course and Infineon winter school.

I am also thankful to my colleagues at the Thomas Johann Seebeck Department of Electronics and Tallinn University of Technology, especially Andres Eke, for encouraging me to take part in summer schools and various teaching assistance opportunities. I would like to thank: Hip Koiv for conducting exciting blood pressure sensor experiments; the TTU satellite design team, for a unique experience; IT manager Fredrick Rang, who was always available and willing to solve any IT-related problems, sometimes accompanying me in several memorable events; staff members Eva Keerov and Jana Rang, for helping throughout the study by providing all information whenever needed; Nodirkhon Yusupov, for demonstrating project management skills and ideas and encouraging me during the summer school; all fellow student members and friends, who have supported me directly or indirectly throughout my studies. I sincerely thank Dr. Vilas Nitnaware and Mandar Jagdale (Mandar dada) for their unwavering support and guidance. I am also thankful to my colleague Miguel E. Flores from Don Bosco University, El Salvador, for collaborating on my research. I want to express a special thank you to my wife, Ketaki, whose unwavering support, boundless patience, and understanding have been a constant source of strength. Through my moments of stress and unavailability, she stood by me with a smile and comforting words. Her prayers and encouragement sustained me throughout this entire Ph.D. journey, and I am forever grateful for her presence in my life. I extend my deepest gratitude to my Guru, teachers, parents, and brother, whose unwavering support, encouragement, and love have been the foundation of my journey. My gurus' and parents' constant presence, guidance, blessings, and belief in me have been a source of immense strength, and I am truly blessed to have had them by my side throughout every step of this path.

Finally, I express my most heartfelt gratitude to the following entities for their financial support during my Ph.D. studies:

- This work partly received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 668995 and is supported by the Estonian Research Council under grants PUT1435, IUT1911, and PRG780.
- The ICT Doctoral School at Tallinn University of Technology.
- The IT Academy Scholarship at Tallinn University of Technology.
- The DoRa Program.

Abstract

Area Efficient Design and Implementation of a Novel Divider Circuit Block

A better electronic system must implement all the essential mathematical operators and indicate that addition, subtraction, multiplication, and division are vital building blocks for modern theories. Division operation is also a derived operation like multiplication; instead of successive addition, it is derived by successive subtraction or multiplication and some controlling conditions. Time, area, and power are the basic requirements of embedded systems, digital systems, integrated circuits, digital circuits, and computer systems. Many techniques which have been designed and implemented to improve the division operation can be classified into digit recurrence and functional iteration classes. Digit recurrence algorithm-based divider techniques are the most commercially implemented dividers, and SRT division is one of the most implemented non-restoring digit recurrence division algorithms. Still, it is restricted to low-order radix due to the requirement of a practically unfeasible quotient selection table. The overlapping region in the quotient selection table can cause a problem in selecting the quotient value. Many performance improvement techniques were researched and used, such as operand scaling, simple circuit staging, overlapping, and pipelined execution, which helped to improve execution time requirements but failed to improve a considerable amount of implementation area requirements. Some studies have also researched and implemented alternative approaches for designing quotient selection logic, such as the Generalized Svoboda algorithm-based divider (GSA), Svoboda-Tung algorithm-based divider (STA), and the new Svoboda-Tung algorithm (NSTA), which only requires a few MSBs of the partial remainder for developing quotient selection logic. Despite using a few MSBs from only a partial remainder to reduce the criticality of quotient selection logic, it has some drawbacks. Overflow can occur due to overcompensation, causing the quotient digit selection from out of the remainder digit range. Because of this, the final remainder value cannot be calculated at the end of the conversion. Thus, the use of such a divider is limited to applications which specifically do not require remainder data.

The implementation of a low area footprint division circuit is needed because of the emerging applications where these devices are used to implement some critical system-on-chip application or improve the existing application; indirect division operation results are not enough. High radix reduces latency but requires a large capacity look-up table, which is impractical for implementation. Furthermore, functional iteration-based dividers use multiplicative algorithms, which are fast but require more significant area requirements for implementation. Functional iteration-based dividers are generally based on series expansion algorithms such as Newton-Raphson, Taylor series, and the Goldschmidt division algorithm, which requires multipliers that add to area overheads. Functional iteration-based dividers generate an error, depending on how close the selected multiple of the reciprocal or anti-divisor value is at the beginning of the initial iteration. Therefore, to reduce rounding-off errors, it needs many anti-divisor values to choose the initial estimation of the quotient approximation, the iterative process that approaches closest to the final value, and convergence to the anti-divisor, i.e., the reciprocal.

Many researchers have worked on various performance improvement techniques, including pre-scaling operands, carry-save remainders, array implementations,

truncations, cascading, and differential LUTs. However, these performance improvement techniques have yet to be fully explored in addressing the research gap for simultaneously utilizing multiple performance improvement techniques with individual input operands. This approach could potentially lead to the development of a new technique or a combination of fast or moderate methods to optimize execution time and implementation area.

This doctoral dissertation's main objective is to design and implement a reduced area divider circuit block with a solution based on the relation between the divisor and the dividend that improves the conversion logic, avoiding rounding off errors and overlapping regions. The design is developed by simulating the proposed technique and cross-verifying it by performing regular sequential and pseudo-random sequential analysis of implementation against standard result tables generated by simulations and the theoretical study of the proposed idea.

Based on the comparative analysis presented, the proposed USP-Awadhoot divider implementation uses 64% less FPGA hardware resources than the Handel-C built-in divider. The variable-latency Quick-Div dividers and fixed-latency radix-n (n = 8, 16) dividers require 5 to 7 times more chip area than the proposed novel USP-Awadhoot divider. The proposed novel USP-Awadhoot divider implementation shows improvements in its FPGA resource utilization in terms of 77% to 88% improvements in the number of required slice logic LUTs (depending on the use of 8-bit or 16-bit operands) and 96% to 96.36% improvements in the number of slice register flip-flops required (depending on the use of 8-bit or 16-bit operands) in Xilinx IP core pipelined divider. This indicates that the proposed novel USP-Awadhoot divider implementation improved FPGA resource utilization.

Similarly, we compared the proposed USP-Awadhoot divider with the LogiCORE IP Divider Generator V4.0 on Xilinx Virtex-6, Virtex-7, Kintex-7, and Spartan-6 FPGAs for a comprehensive evaluation. Xilinx LogiCORE IP Divider Generator V4.0 creates a circuit for integer division based on a non-restoring radix-2 division algorithm or a high-radix division with pre-scaling. The number of slice register flip-flops used in each FPGA IC is constant at 288, although the number of LUTs used changes from 197 to 205, and the number of six input LUT-FF pairs used changes significantly from 215 to 223, for 8-bit implementation in Virtex 7, Kintex 7, Virtex 6, and Spartan 6. Similarly, the number of slice register flip-flops used in each FPGA IC is constant at 3202, although the number of LUTs used changes from 2060 to 2130, and the number of six input LUT-FF pairs used changes significantly from 2185 to 2209, for 32-bit implementation in Virtex 7, Kintex 7, Virtex 6, and Spartan 6. The proposed USP-Awadhoot divider requires 266 to 1836 slice logic LUTs and 146 to 1352 slice register flip-flops.

To summarize, compared to the existing state-of-the-art digit recurrence dividers, the proposed novel USP-Awadhoot divider simultaneously implements multiple performance improvement techniques (i.e., dynamic separate scaling operations) with individual input operands. This approach achieves variable latency and a small area footprint while preventing overlapping regions in the quotient calculation logic.

Lühikokkuvõte

Uudne efektiivne jagamistehte riistvaraline realisatsioon

Kaasaegsetes elektroonilistes süsteemides on vajalik rakendada kõiki põhilisimatemaatilisi tehteid. Liitmine, lahutamine, korrutamine ja jagamine on tänapäevaste rakenduslike algoritmide olulised ehituskivid. Jagamistehe on sarnaselt korrutamisega tuletatud tehe. Kui korrutamine on tuletatud järjestikuse liitmise abil, siis jagamine on tuletatud järjestikuse lahutamise või -korrutamise ja algoritmi spetsiifiliste juhtimistingimuste kaudu.

Aeg, pindala ja võimsus on põhilised nõuded nii manussüsteemidele, digitaalsüsteemidele, integraallülitustele, digitaallülitustele kui arvutisüsteemidele. Enamikku tehnikaid, mis on välja töötatud ja rakendatud jagamistehte täiustamiseks, saab liigitada kahte klassi: numbri korduvkasutusel põhinev ja funktsionaalsel iteratsioonil põhinev klass. Numbri korduvkasutuse algoritmidel põhinevad jagamistehnoloogiad on kõige laialdasemalt kasutatavad kommertsjagurid ning SRT-jagamine on üks enimrakendatud mitte-taastavate numbri korduvkasutuse algoritmide seas. Siiski on selle rakendamine piiratud madala astmearvuga (radix), kuna kõrge astmete arv vajab praktiliselt teostamatut jagatise valiku tabelit.

Jagatise valiku tabeli katvusalad võivad põhjustada probleeme sobiva jagatise väärtuse valikul. Jõudluse parandamise erinevaid tehnikaid on uuritud ja kasutatud, sealhulgas näiteks operandide skaleerimine, lihtne järjestikskeemide kasutamine, kattuvad lahendialad ja jadatäitmine (pipeline execution), mis on aidanud kiirendada algoritmi täitmise aega, kuid pole suutnud oluliselt vähendada realiseerimiseks vajalikku komponentide arvu – seekaudu ka kiibi pindala.

Mõnedel juhtudel on uuritud ja rakendatud ka alternatiivseid lähenemisviise jagatise valiku loogika kavandamiseks, näiteks Generaliseeritud Svoboda algoritmil põhinev jagur (GSA), Svoboda-Tung algoritmil põhinev jagur (STA) ning uus Svoboda-Tung algoritm (NSTA), mis vajavad jagatise valiku loogika loomiseks ainult osa jäägi kõige olulisematest bittidest (MSB-dest). Kuigi jagatise valiku loogika kriitilisust on vähendatud, kasutades ainult mõne MSB väärtusi osalisest jäägist, on sellel realisatsioonil siiski mõningaid puudusi. Ülekompensatsioon võib põhjustada ületäitumist, mille tõttu valitakse jagatise number väljaspool jäägi numbrivahemikku. Selle tulemusena ei ole võimalik teisenduse lõpus jäägi lõplikku väärtust arvutada. Seetõttu on selliste jagurite kasutamine piiratud rakendustega, mis ei nõua jäägiandmeid.

Väikese pindala kasutusega jagamisahela realiseerimine on vajalik uute rakenduste kasutusnõuete tõttu, kus neid seadmeid kasutatakse kas mõne kriitilise süsteemikiibil (SoC) põhineva rakenduse loomiseks või olemasoleva rakenduse täiustamiseks. Sellisel juhul kaudsed jagamistehte tulemused ei ole piisavad. Kõrge astmearv (radix) vähendab latentsust, kuid nõuab mahukat otsingutabelit, mis on praktilise rakenduse seisukohalt ebaotstarbekas.

Lisaks kasutavad funktsionaalsel iteratsioonil põhinevad jagurid multiplikatiivseid algoritme, mis on küll kiired, kuid vajavad suuremat kiibipindala. Sellised jagurid põhinevad tavaliselt jadalahutuse algoritmidel, nagu Newton-Raphsoni meetod, Taylori rida või Goldschmidti jagamisalgoritm, mis eeldavad korrutite kasutamist, suurendades seeläbi pindalanõudeid.

Funktsionaalse iteratsiooni põhised jagurid tekitavad vea, mis sõltub sellest, kui lähedane on valitud pöördarvu (vastandarvu) väärtus tegelikule pöördarvule iteratsiooni alguses. Seetõttu, et vähendada ümardamisvigu, on vaja mitmeid pöördarvu väärtusi,

et valida jagatise lähenduse algväärtus, mis viib iteratiivse protsessi kaudu võimalikult lähedale lõplikule väärtusele ning tagab lähendamise pöördarvule ehk vastandarvule.

Paljud teadlased on töötanud erinevate jõudluse parandamise tehnikate kallal, sealhulgas operandide eelneva skaleerimise, jääkide carry-save meetodi, arvumassiivide teostuste, kärpimiste (truncation), kaskaadimise ja diferentsiaalsete töeväärtustabelite (LUT) kasutamisega. Siiski ei ole neid jõudluse parandamise tehnikaid veel täiel määral uuritud selles kontekstis, kuidas kasutada korraga mitut erinevat tehnikat koos individuaalsete sisendoperandidega. Selline lähenemine võiks viia uue tehnika või kiirete ja mõõdukate meetodite kombinatsiooni väljatöötamiseni, mis optimeeriks täitmisaega ja teostuspindala.

Käesoleva doktoritöö peamine eesmärk on projekteerida ja teostada väikse pindalaga jaguriahel, mis põhineb jagaja ja jagatava vahelisel seosel, parandades teisendusloogikat ning vältides ümardamisvigu ja kattuvaid piirkondi. Lahendus on välja töötatud simuleerides pakutud lahendustehnikat erinevatel platvormidel ning seda on kontrollitud tavalise järjestikuse ja pseudo-juhusliku järjestikuse analüüsi abil, võrreldes tulemusi alternatiivsete algoritmide standardsete simulatsioonitabelite ja teoreetilise käsitluse põhjal saadud tulemustega.

Võrdleva analüüsi põhjal kasutab pakutud USP-Awadhooti jaguri teostus 64% vähem FPGA riistvararesursse kui Handel-C sisseehitatud jagur. Muutliku latentsusega Quick-Div jagurid ja fikseeritud latentsusega radix-n (n = 8, 16) jagurid vajavad 5 kuni 7 korda rohkem kiibipinda kui pakutud uus USP-Awadhooti jagur. Pakutud USP-Awadhooti jaguri teostus näitab olulist paranemist FPGA ressursikasutuses: segment loogika LUT-ide vajadus väheneb 77% kuni 88% (sõltuvalt sellest, kas kasutatakse 8-bitiseid või 16-bitiseid operande) ning segment registri triger (flip-flop'ide) vajadus väheneb 96% kuni 96,36% (jällegi sõltuvalt operandide pikkusest), võrreldes Xilinx IP core järjestikjaguriga. See viitab sellele, et pakutud USP-Awadhooti jaguri teostus parandab oluliselt FPGA ressursi kasutust.

Samamoodi võrreldi USP-Awadhooti jagurit ka LogiCORE IP Divider Generator V4.0 teostusega Xilinx Virtex-6, Virtex-7, Kintex-7 ja Spartan-6 FPGA-de peal, et tagada põhjalik hindamine. Xilinx LogiCORE IP Divider Generator V4.0 loob täisarvude jagamisahela, mis põhineb mitte-taastaval radix-2 jagamisalgoritmil või kõrge radix'iga jagamisel koos eelneva skaleerimisega. Iga FPGA kiibi puhul on kasutatud triger arv konstantne – 288tk. 8-bitise teostuse korral –, kuid LUT-ide arv varieerub 197-st 205-ni ning kuue sisendiga LUT-FF paaride arv muutub märgatavalt 215-st 223-ni Virtex 7, Kintex 7, Virtex 6 ja Spartan 6 kiipide vahel. 32-bitise teostuse korral on triger arv igas FPGA-s konstantne – 3202tk., kuid LUT-ide arv varieerub 2060-st 2130-ni ja kuue sisendiga LUT-FF paaride arv 2185-st 2209-ni. Samas vajab pakutud USP-Awadhooti jagur vaid 266 kuni 1836 segment loogika LUT-i ja 146 kuni 1352 segment registri triger.

Kokkuvõttes, võrreldes olemasolevate tipptasemel numbri korduvkasutusel põhinevate jaguritega, rakendab pakutud uus USP-Awadhooti jagur samaaegselt mitmeid jõudluse parandamise tehnikaid (näiteks dünaamiline eraldi skaleerimine) individuaalsete sisendoperandidega. Selline lähenemine saavutab muutliku latentsuse (mis siiski suures osas vastab või ületab ootusi lahenduskiirusele) ja väikese vajaliku komponentide arvu – mis väljendub kiibi vajaliku pindala vähenemises, vältides samal ajal katvusalade probleemi jagatise arvutusloogikas.

Appendix 1

Publication I

Appeared in:
Patankar, Udayan; Koel, Ants;
"Review of Basic Classes of Dividers Based on Division Algorithm" in IEEE Access, Vol. 9, 23035−23069. DOI: 10.1109/ACCESS.2021.3055735.

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

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. PATANKAR, (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 Licciardo.

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.

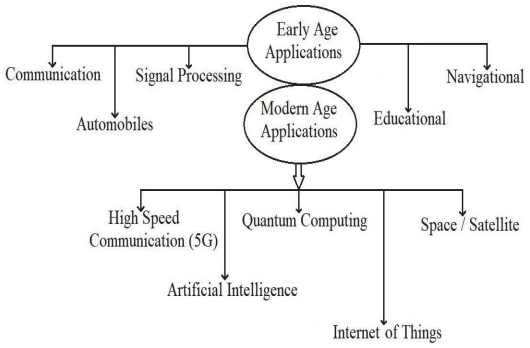


FIGURE 1. The trends of application.

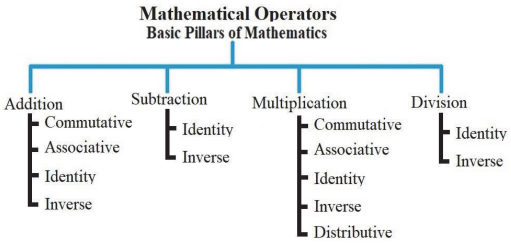


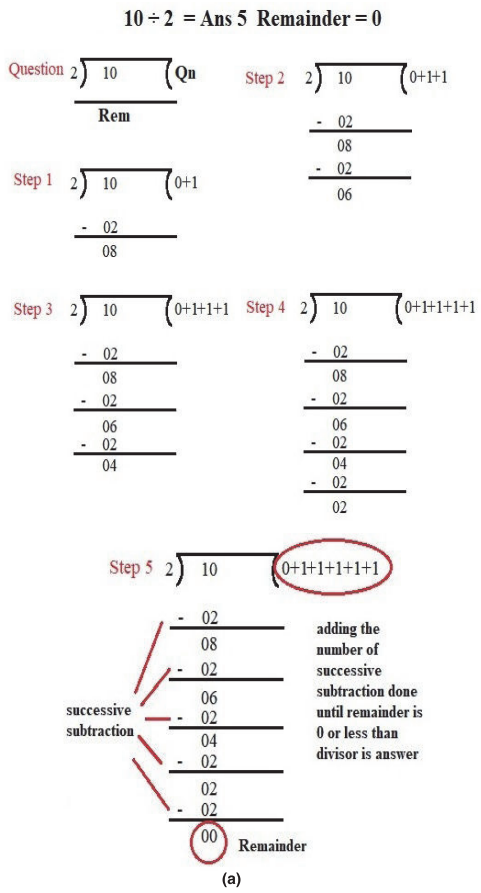
FIGURE 2. The properties of various mathematical operators.

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.



Look up table			
0÷1=0	0÷2=0	0÷3=0	0÷4=0
1÷1=1	2÷2=1	3÷3=1	4÷4=1
2÷1=2	4÷2=2	6÷3=2	8÷4=2
3÷1=3	6÷2=3	9÷3=3	12÷4=3
4÷1=4	8÷2=4	12÷3=4	16÷4=4

(b)

÷	Divisor									
	1	2	3	4	5	6	7	8	9	10
0	0									
1	1									
2	2	1								
3	3		1							
4	4	2		1						
5	5				1					
6	6	3	2			1				
7	7						1			
8	8	4		2				1		
9	9		3						1	
10	10	5			2					1

(DIVIDEND)

(c)

FIGURE 3. (a). The successive subtraction method of the division operation. (b), (c) The look-up table methods of the division operation.

FPGA has implemented many critical systems. It gives the flexibility to implement a system on a chip for different purposes. In FPGA, the arithmetic and logical module (ALM) is an essential building block for implementing the desired logic. FPGA applications are most important and critical for automotive control, online data processing, and a wide range of computational tasks, which can be solved by implementing a small complex system like a computer system on a single chip. All mathematical operations have been implemented electronically, but it is required to focus on improving dividers because the improved technology has given birth to new applications that require a faster speed of response and critical calculation with reduced area requirements; hence more effort has been put into developing improved adders and multipliers instead of improving dividers. This is because of the ease of developing and implementing adder and multiplication logic more effectively than divider logic. The typical latency performance for addition and multiplication falls in the range from a couple of clock cycles to less than ten clock cycles; on the other hand, a division is in the range of tens of clock cycles [14], [73]–[76]. Computer performance could be degraded in the long run due to unimproved divider operation for new computer applications, so better implementation of the divider operation is required. The best way of understanding the merits and demerits of dividers is by classifying division algorithms in different classes. Classes are no more than the indicative name given to a group of algorithms that exhibit similarities in their conversion logic. The hierarchical distribution of various categories of division algorithm is shown in Fig. 4 and is described as follows:

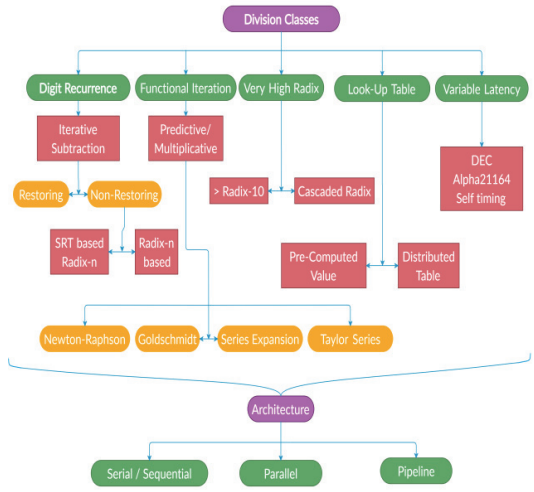


FIGURE 4. The distribution of different division algorithms.

Based on the method of conversion, we can distinguish division algorithms in the following classes.

1. Digit recurrence
2. Functional iteration

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.

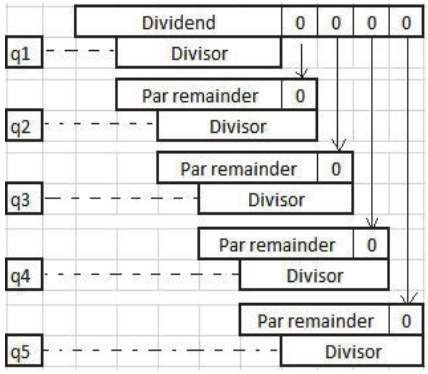


FIGURE 5. Long division algorithm of the digit recurrence class.

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.

$$q_j = 0 \quad \text{if } 2R_{j-1} < D_r \tag{3}$$
$$q_j = 1 \quad \text{if } 2R_{j-1} \geq D_r \tag{4}$$
$$R_j = 2R_{j-1} - q_j \times D_r \tag{5}$$

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.

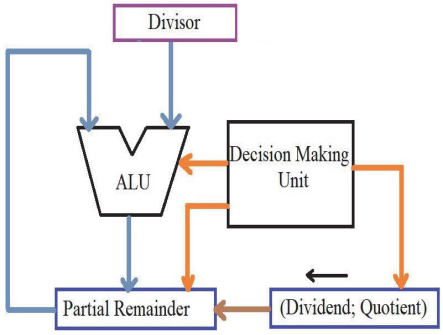


FIGURE 6. The restoring division algorithm of the digit recurrence class.

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:

$$R'_j = 2R_{j-1} - D_r \tag{6}$$
$$q_j = 0 \quad \text{if } R'_j < 0 \tag{7}$$
$$q_j = 1 \quad \text{if } R'_j \geq 0 \tag{8}$$
$$R_j = 2R_{j-1} \quad \text{if } q_j = 0 \tag{9}$$
$$R_j = R'_j \quad \text{if } q_j = 1 \tag{10}$$

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).

$$q_j = -1 \quad \text{if } R_{j-1} < 0 \tag{11}$$
$$q_j = 1 \quad \text{if } R_{j-1} \geq 0 \tag{12}$$
$$R_j = 2R_{j-1} + D_r \quad \text{if } q_j = -1 \tag{13}$$
$$R_j = 2R_{j-1} - D_r \quad \text{if } q_j = +1 \tag{14}$$

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.

$$q_j = -1 \quad \text{if } R_{j-1} < 0 \tag{15}$$
$$q_j = 1 \quad \text{if } R_{j-1} \geq 0 \tag{16}$$
$$R_j = 2R_{j-1} + D_r \quad \text{if } q_j = -1 \tag{17}$$
$$R_j = 2R_{j-1} + \overline{D}_r + 1 \quad \text{if } q_j = +1 \tag{18}$$

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 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.

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.

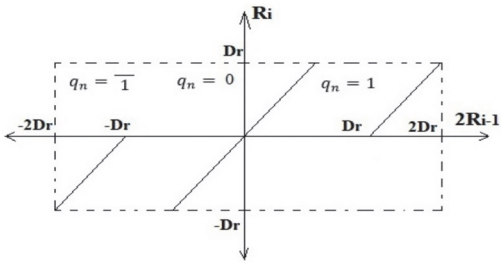


FIGURE 7. The radix-2 SRT algorithm.

$$q_j = \overline{1} \quad \text{if } 2R_{j-1} < -D_r \tag{19}$$
$$q_j = 0 \quad \text{if } -D_r \leq 2R_{j-1} \leq D_r \tag{20}$$
$$q_j = 1 \quad \text{if } 2R_{j-1} \geq D_r \tag{21}$$

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

$$\frac{1}{2}(n-1) \leq m \leq n-1 \tag{22}$$
$$n = 2^b \quad \textit{and } k = {}^{x}/_{b} \tag{23}$$
$$Q = \sum_{j=1}^{k} q_j n^{-j} \tag{24}$$

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].

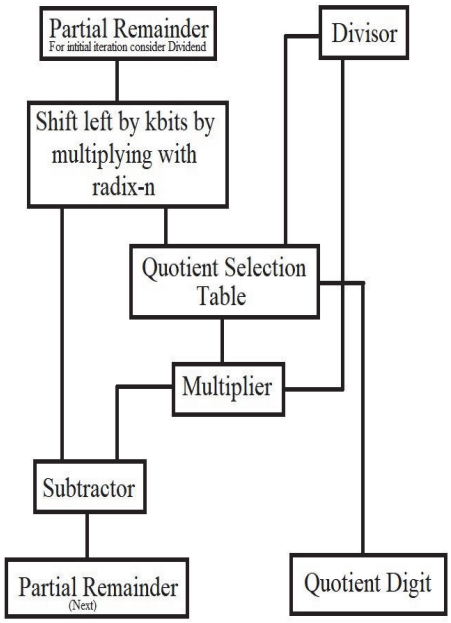


FIGURE 8. Block diagram of SRT algorithm.

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...$-$1, 0, +1... m$-$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 express as (25)

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

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 slop $\rho + k$ and $-\rho + k$ [14], [21], [58], [62], [64]–[68].

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

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

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

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

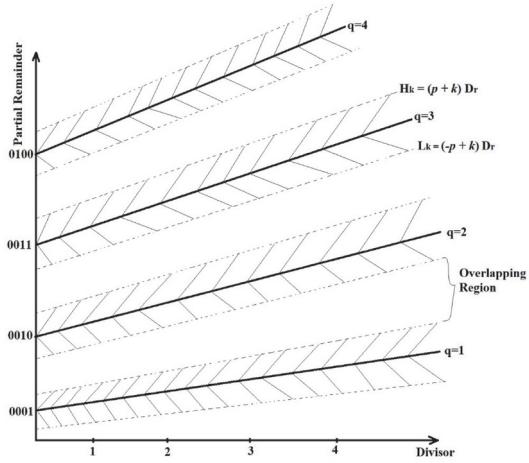


FIGURE 9. Partial remainder – Divisor (PD) graph.

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

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

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.

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.	******

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 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.

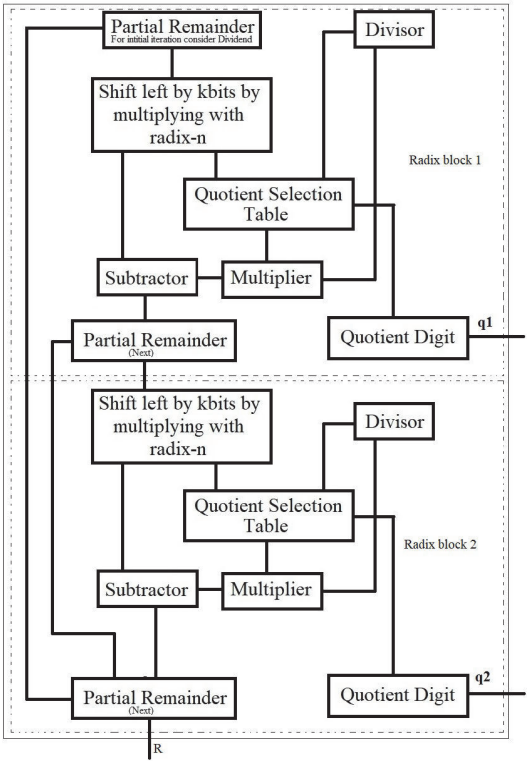


FIGURE 10. Block diagram of cascaded implementation of SRT algorithm.

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].

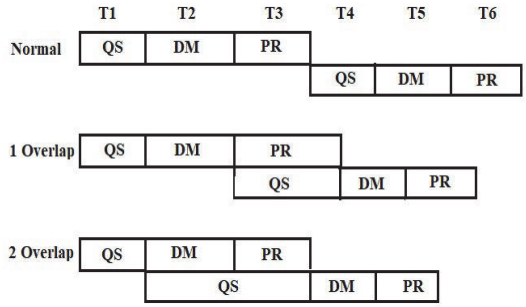


FIGURE 11. The conditions of execution.

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 decurrence 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 > 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

$$Range = \{0, \pm 1, \ldots\ldots, \pm m\} \tag{29}$$

$$Boundry\ limit = \left\{n/2 + 1 \leq m \leq n - 1\right\} \tag{30}$$

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)

$$\left\{-m/n - 1 < R_j < m/n - 1\right\} \tag{31}$$

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

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

$$Range = \{0, \pm 1, \ldots\ldots, \pm m\} \tag{33}$$

$$Boundry\ limit = \left\{n/2 + 1 \leq m \leq n - 1\right\} \tag{34}$$

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 Rj 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 (Dr) is given as (36)

$$SBD = \{-1 \leq m \leq 1\} \tag{35}$$

$$D_r range = \{0, 1, \ldots\ldots, n - 1\} \tag{36}$$

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

$$Boundry\ limit = \left\{-m/n - 1 < R_{j+1} < m/n - 1\right\} \tag{37}$$

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 .... 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).

$$P(a) = X_0 + X_1 a + X_2 a^2 + X_3 a^3 \tag{38}$$

The initial order coefficients X0 and X1 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].

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

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

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

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

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

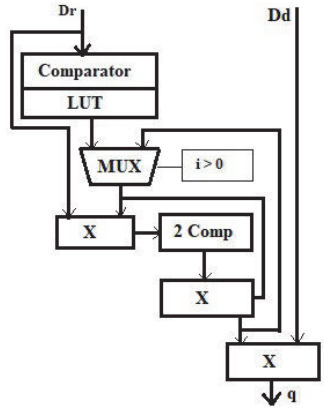


FIGURE 12. The block diagram of Newton–Raphson algorithm implementation.

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.

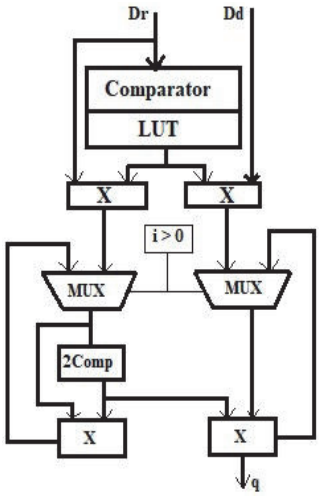


FIGURE 13. The block diagram of Goldschmidt algorithm implementation.

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].

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

$$q = {}^{a}/_{b} = a \times g(y) \quad (45)$$

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,

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

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

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

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 1then (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.

$$q = {}^{D_d}/_{D_r} \quad and\ X_0 = {}^{1}/_{D_r} \tag{52}$$

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

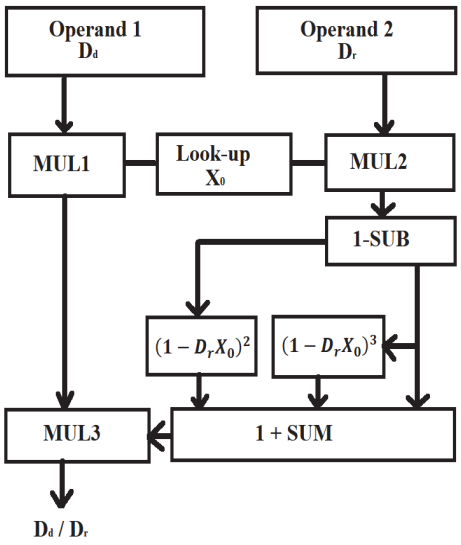


FIGURE 14. Operational block diagram of Taylor series algorithm.

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 essentializes 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

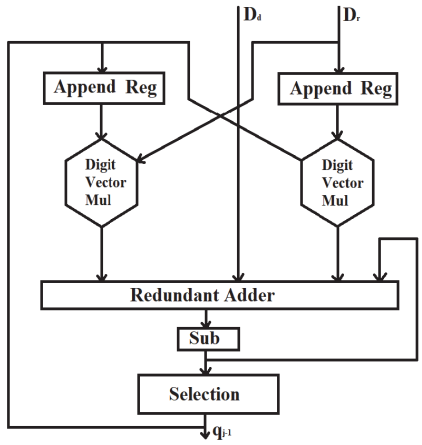


FIGURE 15. Operational block diagram of the online division algorithm.

easied 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 enhancive 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.

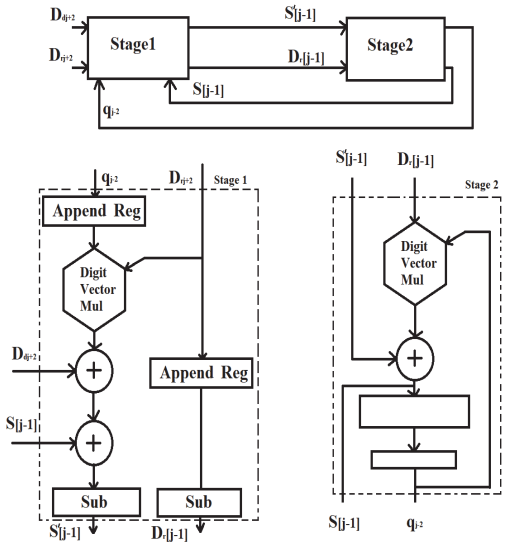


FIGURE 16. Block-level implementation of 2-stage pipelined online divider.

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 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

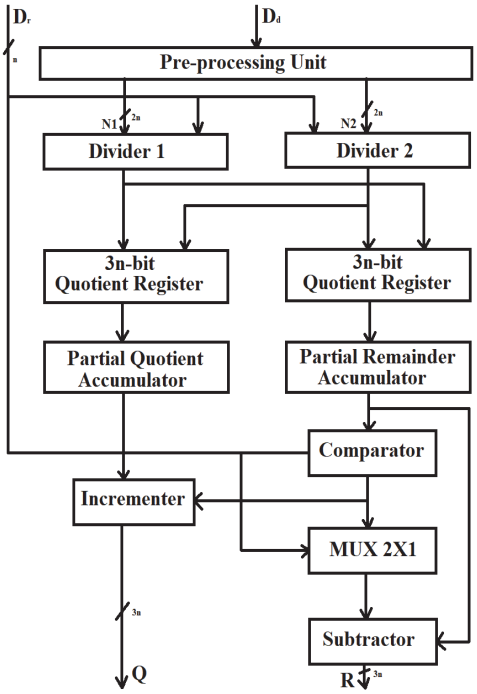


FIGURE 17. Block-level implementation of small dividend division algorithm.

- 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

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

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

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

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

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 $D_{d1}, D_{d2}, D_{d3} \ldots\ldots\ldots D_{dk}$. D_{dupk} is the upper part of the dividend $D_{dk+1}, D_{dk+2}, \ldots\ldots\ldots D_{dm}$. 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

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

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

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 (XCVU9P-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 and type. Due to this, Quick-Div requires additional 3 cycles for sign conversion, as mentioned in Table 5.

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

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 DL remaining, i.e., n-p bits.
- Perform Taylor expansion to the above point, and we will get $Y = N/D_K\ (1\text{-}D_L/D_K + D_L^2/D_K^2 - D_L^3/D_K^3 + \ldots)$.
- 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.

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

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 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.

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

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 1to 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-Rapsom 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 fussed 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 NVIDIAs 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 -Dr to +Dr 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 $q_j = \overline{1}$ 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, ….., -1, 0, 1, ……., m-1, m} The algorithm implements 2's complement value of Dr instead of –Dr, 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 is the same as the SRT algorithm, with the only difference is that it retires more than ten quotient bits in one iteration; it requires a very large look-up table with a big capacity for quotient selection logic 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	$\left\{\frac{mn}{(m+1)(n-1)} < D_r < \frac{m(n-2)}{(n-1)(m-1)}\right\}$ $\{-m/n-1 < R_j < m/n-1\}$ $Range = \{0, \pm 1, \ldots\ldots, \pm m\}$ $Boundry\ limit = \{n/2+1 \leq m \leq n-1\}$	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 quotient digit out of the radix range as an overflow occurred due to compensation
			It requires pre-scaled operands and can work on conventional and signed digit number range
			Like the SRT algorithm, it is also a radix-n based algorithm
7	New Svoboda-Tung Algorithm	$SBD = \{-1 \leq m \leq 1\}$ $D_r range = \{0,1, \ldots\ldots, n-1\}$ $Boundry\ limit = \{-m/n-1 < R_{j+1} < m/n-1\}$	It records/re-stores the partial remainder's two most significant digits to overcome overflow due to compensation
			This arrangement allows for addition/subtraction with carry propagation up to one left position
			Rest the working is similar to that of the Svoboda-Tung algorithm
			Total conversion time is given as a collective time required for scaling, recursion, conversion in terms of an initial clock cycle, thus $T_{\text{div}} = (T_{scale} + T_{rec} + T_{Conv}) * T_{clk}$
			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
			The major drawback of this algorithm is high hardware overhead
8	Look-up table	******	It stores the values, logic, numbers, quotients, etc., which are useful to execute division
			Look-up table class algorithm can be utilized along with functional iterative class and high radix algorithms
			Look-up tables are useful in initial approximation in the case of SRT and functional iteration class algorithms.
			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
9	Newton-Raphson	$Q = D_d/D_r = p \times (q)^{-1}$ $f(X) = 1/X - q^{-1} = 0$ $X_{i+1} = X_i - \frac{f(X_i)}{f'(X_i)}$ $X_{i+1} = X_i - \frac{(1/X_i - q^{-1})}{1/X_i^2} = X_i \times (2 - q^{-1} \times X_i)$ $\epsilon_{i+1} = \epsilon_i^2\ (q^{-1})$	It comes under the functional iteration class
			Requires look-up table
			It works on the estimation technique
			It considers the convergence of quotient by estimation or prediction
			The final quotient is derived by multiplying approximated reciprocal and dividend
			Shows error due to inaccuracy of quotient digit prediction or estimation
			It requires multiplication and addition or subtraction at each iteration
			The accuracy can be improved by selecting a proper root at the beginning
			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
			Multiplier, quotient select look-up table, and control logic gives the approximate area requirement for algorithm implementation

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

10	Series Expansion	$g(y) = g(p) + (y-p)g'(p) + \frac{(y-p)^2}{2!} g''(p) + \cdots$ $q = {}^{a}/_{b} = a \times g(y)$	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	$q = {}^{D_d}/_{D_r}$ and $X_0 = {}^{1}/_{D_r}$ $q = D_d X_0 \{1 + (1-D_r X_0) + (1-D_r X_0)^2 + (1-D_r X_0)^3\}$	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	$D_d = d * Q$ $D_d = D_{dupk} n^k + D_k$ $b_k = (-n^{-k} D_{dk}) \bmod d$ $b_k = (-n_{mod\ d}^{-k} D_{dk}) \bmod d$	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 equaled 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 equaled 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	Approximatly 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	Area utilization of implementation is approximately equaled to the area required to implement quotient select table and carry-save adder (CSA)
		Demand to use very large look up tables			
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	$\left\{\left\lceil\log_2\frac{x}{f}\right\rceil + 1\right\} t_{mul} + 2$ $if\ t_{mul} > 1$ $\left\{2\left\lceil\log_2\frac{x}{f}\right\rceil + 3\right\}$ $if\ t_{mul} = 1$ and approximately 16-18 cycles	Area utilization of implementation is approximately equaled 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			
	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			
	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 equaled 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	$T_{\text{div}} = (T_{scale} + T_{rec} + T_{Conv}) * T_{clk}$.	******	Area utilization of implementation is approximately equaled 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	$T_{div} = \left(3 + \left(\frac{W}{2}\right) + T_{Conv}\right) * \left(22 + \frac{W}{4}\right)$	******	Area requirement is appriximaly 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	Approximatly time required to access quotient select table, multiple form, and subtraction	$\left\{\left\lfloor\frac{x}{n}\right\rfloor + Scale\right\}$ 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	Approximatly time required to access two serial multiplication and subtraction	$\left\{2\left\lfloor\log_2\frac{x}{j}\right\rfloor + 1\right\} t_{mul} + 1$ 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	Approximatly time required to perform one-two multiplication	$\left\{\left\lfloor\log_2\frac{x}{j}\right\rfloor + 1\right\} t_{mul} + 2 \;\; if\; t_{mul} > 1$ $\left\{2\left\lfloor\log_2\frac{x}{j}\right\rfloor + 3\right\} \quad if\; t_{mul} = 1$ 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 prescaling 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 studyof 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. Ercegovac 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 160-ns 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 multicore 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.

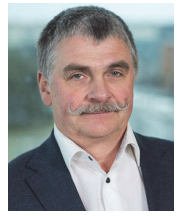
[55] A. Thall, "Extended-precision floating-point numbers for GPU computation," in *Proc. Special Interest Group Comput. Graph. Interact. Techn. Conf.*, Boston MA, USA, Jul. 2006, p. 52.

[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. (ICESS)*, 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 Svobota-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. Guyo, "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. 74440O, doi: 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.
[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.

Appendix 2

Publication II

Appeared in:
Patankar, Udayan; Koel, Ants; Patankar, Sunil; Flores, Miguel;
"Area Efficient Hexadecimal Divider Circuit Implementation Based on USP-Awadhoot Division Algorithm" in 2021 IEEE International Conference on Engineering, Technology, and Innovation (ICE/ITMC), 2021, pp. 1–8, doi: 10.1109/ICE/ITMC52061.2021.9570263.

Area Efficient Hexadecimal Divider Circuit Implementation Based on USP-Awadhoot Division Algorithm

1st Udayan Patankar
TJS Department of Electronics
TalTech Tallinn University of Technology
Tallinn, Estonia
udayan.patankar45@gmail.com
Orcid Id- 0000-0003-4167-6755

2nd Dr. Ants Koel
TJS Department of Electronics
TalTech Tallinn University of Technology
Tallinn, Estonia
ants.koel@taltech.ee
Orcid Id- 0000-0001-5635-0139

3rd Sunil Patankar
Kavi Kulguru Kalidas Sanskrit University
Ramtek, India
smpatankar1956@yahoo.com

4th Miguel E. Flores
School of Electronics
Don Bosco University
Soyapango, El Salvador
miguel.flores@udb.edu.sv
Orcid Id- 0000-0003-0514-6239

2021 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC) | 978-1-6654-4963-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICE/ITMC52061.2021.9570263

***Abstract*— Arithmetic operations are crucial in the current age of technology and development. Electronic implementation of mathematical operations is a very important and critical part of a digital system. Even though the number of transistors on a chip, increases beyond Moore's law prediction, it is quite complicated to implement arithmetical operations; a sophisticated algorithm is essential to successful implementation. Different dividers have been developed based on various algorithms to provide better results for division implementation in digital hardware. Generally, the major aspects considered are quotient convergence rate, hardware primitives, area overheads, and mathematical formulation. Many algorithms were implemented to achieve dividers, which provide a solution with or without any error percentage. Although many error-resilient applications exist, such dividers fail to work satisfactorily in terms of power consumption and overall circuit performance. In some cases, estimation and approximation-based dividers are used to provide faster usage with a considerable error, but digit recurrence algorithms are used in most commercial applications, which requires more area. Thus, we developed a new, three-stage zero error digit recurrence USP-Awadhoot division algorithm. The present article illustrates the design and implementation statistics of an area-efficient divider circuit based on a novel USP-Awadhoot division algorithm and the effective use of a hexadecimal number system to achieve ease in quotient conversion logic.**

Keywords*— *Approximate computing, Dividers, Functional iteration, Prediction, FPGA, SRT

I. INTRODUCTION

It is perverse to consider mathematics as terminology, which deals with, numbers, whereas mathematics itself governs the very deep meaning of science of numbers and their relations and sometimes both. Theoretical mathematics allows us to see the world in terms of pure equations, relations and provides studies to develop various mathematical theories. In the past, before the beginning of the computer and electronics era, many researchers had given different mathematical theories to support human evaluation. However, such theories may not be implemented at that time. Currently, in the era of the electronics and computer revolution, another side of mathematics has been developed, providing the implementation for various mathematical theories that are useful for improving human beings' lives. Collectively, we know this other side of mathematics as applied mathematics. It gave a more significant aspect to mathematics, applying its concepts in human life.

The initial phases of industrialization were reliant on the new ways of theoretical mathematics and physics in the industries to develop mass-production techniques that could provide a better solution to economic difficulties when producing various items or products. These applied physics and mathematics efforts gave birth to new possibilities, leading to the newborn field of electronics and integrated circuits, which has proven valuable and innovative for existing applications like communication, transport, and calculations. In the current computer generation, 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, high-performance graphics rendering system like graphic processing units (GPU), complex systems on chips, central processing unit, biomedical equipment, fuzzy control, space engineering.

An increase in the applications increases the demand for the implementation of various arithmetic operations such as addition, subtraction, multiplication, and division with a more sophisticated approach. A better electronic system needs to implement all the basic mathematical operators' basic properties, as illustrated in Fig. 1. It indicates that addition, subtraction, multiplication, and division stand as vital building blocks of implementing modern theories of theoretical and applied mathematics [2]. Like the multiplication operation, division operation is also a derived operation, where instead of successive addition, it involves successive subtractions and critical controlling conditions, which indicates that the division operation is a difficult activity for digital hardware. The involvement of successive subtraction makes it highly dependent on the order of two quantities connected by the division operator.

This project has received funding from the Estonian Research Council Institutional Research, EAS - Enterprise Estonia, and partly from the European Union's Horizon 2020 Research and Innovation Program.

More efforts have been put into improving addition, subtraction, and multiplication operators, than the efforts put forward for performing a division operation. The upcoming working application areas of high-speed computation, embedded systems, artificial intelligence [1] [3] [7-8], complex SOC [9], vision systems [1-2] [5-6], automotive control [9], telecommunications [1], the internet of things (IoT), cryptography [1] [4] and many others give the possibility of further improvements in the division operation's implementation.

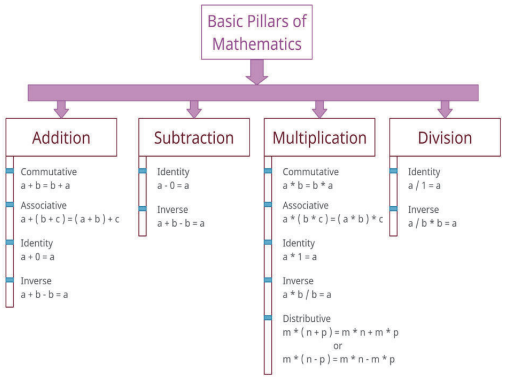


Fig.1. Basic pillars of mathematics

Algorithms are the best tool for implementing derived operators like division, giving a systematic way of performing various critical controlling conditions involved along with division operation. Area, time, and power consumption are the three topics of interest from the system implementation point of view. This gives us the motivation to design a new algorithm that provides improved area utilization and moderate latency. The implementation of the USP-Awadhoot division algorithm has shown advantages over current alternatives, those advantages are presented and compared with current options for division implementation.

II. Existing Theories & Previous Work

At the beginning of the electronics era, mathematical operations, i.e., addition subtraction, multiplication, and division, were involved in CPUs, microprocessors, calculators, counting equipment, and devices in applications with low computation requirements. Division operation was performed based on sequential, linear operation, and digital circuitry to express logic function with high accuracy on account of large area and latency [10-12]. The applications developed at this stage required area improvement for the division because its current implementations lack in area and latency efficiency, and it was less expected to occur in normal working conditions [13-15].

The distributive property of multiplication and the commutative, associative, identity, and inverse properties like addition make it easier for improvements in the final implementation [16-25]. Thus, many efforts were put into improving adders and multipliers to improve system performance in terms of area and latency; the one-step multiplier algorithm from Wallace [26] is an example. Several options were suggested and implemented for performing division operation, which can be presented in different classes: digit recurrence, functional iteration, very high radix, a look-up table, and variable latency class divider.

A comparative study of various dividers from the above classes is presented in Table I; it explains the basic logic of computation and the architecture used, which points out the pros and cons of the particular class algorithms used for division circuit implementation. It gives basic points of comparison with the proposed algorithm implementation. In the early years of electronics, the use of division was very much visible in computer architecture, CPUs, ASIC implementation, FPGA, embedded systems. Division operation is one of the most resources demanding operations in the arithmetic and logical unit (ALU) and for graphic processing unit (GPU) work.

Division operation has consecutive sort of operations with the most exorbitant requirements as far as computational intricacy than other numerical operations. To cater to such application needs, various dividers based on diverse algorithms, like the digit recurrence, functional iteration, high radix, and look-up table divider, either in combination or alone, have been discussed and implemented in the past. When we discuss the various division algorithm's implementation statistics, it is preferable to classify them into different classes based on common properties. Classes are no more than the indicative name given to a group of algorithms that exhibit similarities in their conversion logic and the physical hardware arrangement. The hierarchical distribution of various classes of division is described based on the four factors representing conversion logic, hardware architecture performance, and execution type as follows:

Based on the method of conversion, we can distinguish division algorithms in the following classes.

1. Digit recurrence
2. Functional iteration
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. Predictive type / Multiplicative type

Even though we distribute different implementations of a division operation, it shows some interdependency as well. Meaning the classes are the indicative term used to highlight their important part of working and implementation. It shows that a divider from one class can be implemented fully or partially based on another class; for example, a digit recurrence divider can be implemented in serial or sequential type or pipelined type.

TABLE I
COMPARATIVE STUDY OF DIFFERENT DIVISION ALGORITHMS

Sr. No.	Algorithm	Equations	Important Points
1	SRT	For J^{th} iteration $q_j = \overline{1}$ if $2R_{j-1} < -D_r$ $q_j = 0$ if $-D_r \le 2R_{j-1} \le D_r$ $q_j = 1$ if $2R_{j-1} \ge D_r$ has one of the values -m, -m+1… -1, 0, +1. . . m-1, m, where m is an integer comprises k digits of radix-n as $\frac{1}{2}(n-1) \le m \le n-1$ $n = 2^b$ and $k = {}^{x}/_{b}$ $Q = \sum_{j=1}^{k} q_j n^{-j}$ Quotient q is generated as a division of the dividend by a divisor of x most significand bits retires b bits of the quotient in each iteration Thus, it is called a radix-n performing k iterations to get desired quotient.	It is a 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, ….., -1, 0, 1, ….., m-1, m}
			The algorithm implements 2's complement value of Dr instead of –Dr, 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
2	Very high radix	*******	It is the same as the SRT algorithm, with the only difference that it retires more than ten quotient bits in one iteration; it requires a very large look-up table with a big capacity for quotient selection logic
			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
3	Taylor Series	$q = {}^{D_d}/_{D_r}$ and $X_0 = {}^{1}/_{D_r}$ $q = D_d X_0 \left\{ \begin{matrix} 1 + (1 - D_r X_0) + (1 - D_r X_0)^2 \\ +(1 - D_r X_0)^3 \end{matrix} \right\}$ D_d = Dividend and D_r = Divisor $1/D_r = Antidivisor$	It is 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
4	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
5	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 = Dividend and $(q)^{-1} = Antidivisor$	The accuracy can be improved by selecting a proper root at the beginning
			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 multiplications
			Multiplier, quotient select look-up table, and control logic gives the approximate area requirement for algorithm implementation
			The final quotient is derived by multiplying approximated reciprocal and dividend
			Shows error due to inaccuracy of quotient digit prediction or estimation
			It requires multiplication and addition or subtraction at each iteration
6	Svoboda Algorithm And Svoboda-Tung Algorithm	$\left\{ \frac{mn}{(m+1)(n-1)} < D_r < \frac{m(n-2)}{(n-1)(m-1)} \right\}$ $\{-m/n - 1 < R_j < m/n - 1\}$ $Range = \{0, \pm 1, \ldots\ldots, \pm m\}$ $Boundry\ limit = \{n/2 + 1 \le m \le n - 1\}$ $m = Range\ of\ SBD\ and\ n = Radix$	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 quotient digit out of the radix range as an overflow occurred due to compensation
			It requires pre-scaled operands and can work on conventional and signed digit number range
			Like the SRT algorithm, it is also a radix-n based algorithm with sign binary digit numbers

978-1-6654-4963-2/21/$31.00@2021 IEEE

Digit recurrence algorithm-based dividers are the most commercially implemented dividers due to ease in implementation because of their relatively less complex conversion logic. The error is reduced by retiring a fixed number of quotient digits by incremental operand use in the digit recurrence method. Restoring and non-restoring are the two subtypes of the digit recurrence division algorithm. The restoring division algorithm replicates the long division method, which resembles a paper and pencil algorithm, which generally showcases the application of iterative subtraction. Having an easy and less complex conversion logic for the quotient is a merit, but it exhibits relatively higher latencies as a demerit. A non-restoring algorithm type is very similar to that of restoring algorithm, except that it is not required to restore the partial remainder.

SRT division is one of the most implemented non-restoring digit recurrence division algorithms. It was developed individually by three researchers Sweeney, Robertson, and Tocher. Who proposed utilization of 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]. Although SRT dividers exhibit simple conversion logic, the implementation of the SRT divider is restricted to low order radix due to the practically unfeasible quotient selection table requirement. A rise in radix value increases the size of the quotient selection logic table beyond practical limits of implementation, causing area overhead. High-radix division algorithms are implemented with different architectures like array structure and cascading architecture but require a comparatively higher area. Usually, Look-up table-based dividers are mostly used partially combined with other types of dividers like SRT, High radix type [11, 18]. Svoboda and Svoboda-Tung algorithm is also a radix base algorithm but requires only partial remainder MSB's to decide quotient result.

The basic terminology used in all of the above-stated implementations is similar to the generalized paper and pencil method but inefficient in terms of area and latency, but ultimate quotient results are more accurate. After digit recurrence class dividers, functional iterative class dividers come second. The functional division class showcases the use of iterative multiplication over the estimation or prediction primary reciprocal root for the quotient value based on the chosen priming function. The effective implementation of such dividers is dependent on the accuracy in selecting the nearest reciprocal root of the priming function out of the number of reciprocal roots. The main disadvantage of using functional iteration is the criticality in obtaining the exact result after rounding off, leading to rounding error in the divider's final quotient result. 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 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. Variable latency division implementation is very difficult due to synchronization problems. The extra circuitry required for synchronizing and speculation of quotient digit cost more area. Thus, more focus was expressed to improve adders and multipliers, requiring 2 to 6 cycles, whereas the divider requires a 4 to 80 cycle depending on the combination of architecture and algorithm or technique. As explained in [13, 18], it is vital to improving divider implementation in terms of area and latency, along with adder and multipliers. To improve the system performance of electronic and embedded systems, it is required to reduce area requirements for divider implementation. The digit recurrence algorithm can be appropriate for applications such as embedded systems, FPGA design, ASIC design where minimizing chip area is the priority, [18], latency can be high. The SRT division algorithm is the most famous and widely implemented by commercial applications, but the increase in quotient conversion logic complexity with an increase in radix causes it to be practically unimplementable.

III. METHODS

a. Presented study, Research question & Hypothesis

TABLE II
COMPARISON TABLE

Name of algorithm	Latency (Cycle)	Approx. area
SRT	$\left\{\left\lceil\frac{x}{n}\right\rceil + Scale\right\}$	Quotient selection table + CSA
Svoboda and Svoboda-Tung	*****	LUT+1 full word length fast carry propagation adder+ 2 full word length carry free adder/subtractor +2 full word length latches
New Svoboda-Tung	*****	LUT+1 full word length fast carry propagation adder+ 2 full word length carry free adder/subtractor +2 full word length latches
Very high radix	$\left\{\left\lceil\frac{x}{n}\right\rceil + Scale\right\}$	Quotient select table + CSA
Taylor Series	8-12 cycles	LUT+three full word length multipliers
Newton-Raphson	$\left\{2\left\lfloor\log_2\frac{x}{j}\right\rfloor + 1\right\} t_{mul} + 1$	One multiplier+ table +control
Goldschmidt	$\left\{\left\lfloor\log_2\frac{x}{j}\right\rfloor + 1\right\} t_{mul} + 2 \;\; if\; t_{mul} > 1$ $\left\{2\left\lfloor\log_2\frac{x}{j}\right\rfloor + 3\right\} \quad if\; t_{mul} = 1$ 16-18 cycles	One full word length multipliers+ adders+ LUT

Table II shows the comparative study of implementations of various division algorithms in terms of area and latency. As per the study of previously implemented division algorithms, it is clear that the restoring and some non-restoring algorithms implement simple conversion logic but require a long conversion time and a large area for practical implementation. However, the SRT algorithm is the first choice for commercial implementation in most soft and modern processors like Intel's Pentium processor, FPGA

978-1-6654-4963-2/21/$31.00@2021 IEEE

controllers, and ALU units of complex hardware. It is restricted to certain low radix values, especially less than 10, due to an increase in the quotient selection logic's criticality and the enormous increase in area requirements for storing look-up tables. 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, restricting its use with applications where the unscaled remainder is a must. The main disadvantage of using functional iteration is the criticality in obtaining the exact result after rounding off, leading to rounding error in the divider's final quotient result. Based on this study, we derive the need for designing an area-efficient division algorithm.

b. Applied Research Methods- USP – Awadhoot division algorithm

As per the study shown in previous sections, it is clear that digit recurrence algorithms are most suitable for embedded and consumer applications where cost and area are of utmost importance rather than latency time. To improve area efficiency in digit recurrence dividers, we propose a novel state-of-the-art USP-Awadhoot division algorithm. USP – Awadhoot division algorithm is conceptualized based on the restoring type of digit recurrence concept of the division algorithm, one of the famous and most commercially implemented class of division algorithms.

Unlike a non-restoring type SRT division algorithm, the USP-Awadhoot algorithm is a restoring type of algorithm and more area efficient. We developed a new algorithm consisting of three stages during our research, i.e., stage1, stage 2, and stage 3, and named it as USP-Awadhoot division algorithm, as shown in Fig. 2. Preliminary, the working of the USP-Awadhoot division algorithm is described in stage 1 as a pre-processing input circuit stage, stage 2 as the main processing circuit stage, and stage 3 as the post-processing circuit stage. D_d represents the dividend value, D_r represents the divisor value, Q_n represents the quotient value and R_{em} represents the remainder value of the proposed USP-Awadhoot division algorithm. The key point in using restoring type digit recurrence divider is that it requires a combination of simple operations like addition, shifting, multiplication as shown in (1), and the remainder has to fulfill the requirement stated in (2).

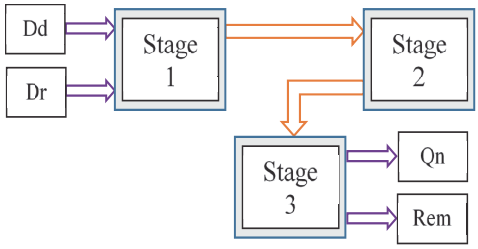


Fig.2. Block diagram of USP-Awadhoot division algorithm

$$Dividend\ (D_d) = \{Quotient(Q_n) \times Divisor(D_r)\} + Remainder(R_{em}) \quad (1)$$

$$0 \leq Remainder(R_{em}) \leq Divisor(D_r) \quad (2)$$

Fig. 3 illustrates the functional block diagram of the USP-Awadhoot division algorithm. Operations and responsibilities of stage 1 of the proposed division algorithm are named pre-processing circuit stage, further sub-categorized into state 101,102,103, and 104. State 101 is responsible for collecting and preserving a copy of input operand data (D_d and D_r) in case, the occurrence of restoring the previous condition. State 102 and 103 are responsible for normalizing input and supplying it in the required hexadecimal format to further state 104. State 104 is responsible for pre-processing normalized hexadecimal operand data (D_d and D_r) and supply required data for the Awadhoot matrix. Therefore, operations and responsibilities of stage 2 of the proposed division algorithm are further sub-categorized into states 105, 106, and 107.

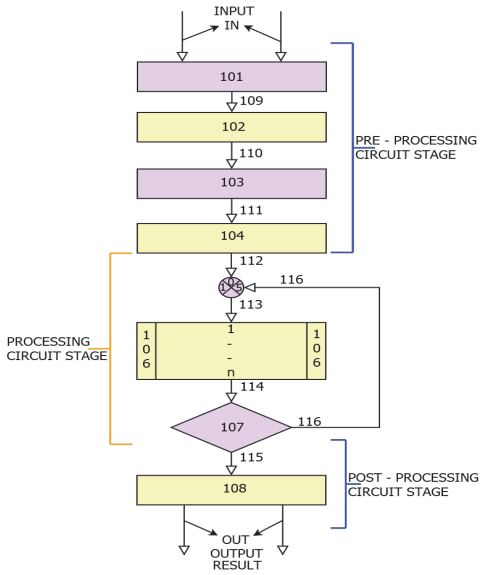


Fig.3. Functional block diagram of the USP-Awadhoot division

State 105 represents the restoring stage in case of the occurrence of restoring the previous condition, and in a normal case, it passes previous iteration data from one iteration to the next iteration of the Awadhoot matrix. State 106 and 107 represent the main processing circuit stage of the Awadhoot matrix. Later, state 107 is responsible for providing group quotient and remainder value from every iteration of the Awadhoot matrix to the next state 108. The operations and responsibilities of stage 3 of the proposed division algorithm represented state 108 and considered a post-processing circuit stage.

Awadhoot matrix is the tabular arrangement of presenting pre-processed input for main iterative circuit stages. Similar to the SRT algorithm, this algorithm also generates the quotient bits separately in different iterations. All these quotients are generated separately, and remainder values are fed to the recombination circuit in the post-processing circuit stage. The final quotient value and remainder value are generated at the end of the post-processing stage at state 108 and available for further display or transmission to different devices if needed. The hypothesis is that the USP-Awadhoot division algorithm can reduce resources used and improve division calculation speed compared with current

978-1-6654-4963-2/21/$31.00@2021 IEEE

implementations of other division algorithms. The implementation of the proposed algorithm was used for validation of its performance and evaluation of its results. Outputs (Q_n and R_{em}) obtained were compared with their truth table answers and verified the hardware resources utilization to understand the hardware requirements for the proposed USP-Awadhoot division algorithm implementation.

c. Research Model & Instrument

For implementing the proposed USP-Awadhoot division algorithm, we have used the Vivado 2016 simulation tool with the Zybo development board based on Xilinx Zynq XC7Z010. We also tested the implementation of the proposed USP-Awadhoot division algorithm on Quartus prime lite simulation software with Cyclone IV development board based on EP4CE6E22C8N Cyclone IV FPGA manufactured by Altera to generate a truth table and cross-verify simulation results by comparing output result. Table 3 represents the resource utilization of the divider circuit implemented based on the USP-Awadhoot division algorithm. Figure 4 and 5 represents the comparative analysis of proposed divider based on Awadhoot division algorithm with different SRT digit recurrence algorithm-based radix divider, different digit recurrence division algorithms based dividers and other division algorithm class dividers.

d. Experimental set-up

To test the implementation of the proposed USP-Awadhoot division algorithm, we prepared the HDL code based on the functional block diagram of the proposed USP-Awadhoot division algorithm as shown in figure 3 and programmed it into the Zybo development board based on Xilinx Zynq XC7Z010 and Cyclone IV development board based on EP4CE6E22C8N FPGA manufactured by Altera. All the possible input operand conditions are checked sequentially and randomly on both experimental test boards. To test the working of implementation, we used a pseudorandom code generator to get random values for different input operands as well as we tested the working of proposed algorithm implementation with static inputs supplied via separate single pole single throw rocker switches. To determine the working latency and conversion speed, we have executed simulations with different clock frequencies and tested them by implementing them into the Zybo development board based on Xilinx Zynq XC7Z010 and Cyclone IV development board based on EP4CE6E22C8N FPGA manufactured by Altera.

IV. FINDINGS

a. Data Collection and Analysis

During various simulations performed on the Vivado 2016 simulation tool and Quartus prime lite simulation software, the results of various input operand combinations are collected, forming a truth table for all possible combinations of input operands. The collected simulation data is summarized into time, current operands (dividend and divisor) value, the present value on control signals, and the outputs (quotient, remainder, error state, and valid output) obtained by the computation of the algorithm. Data collected from the output of hardware implementation were compared with their know solutions from the reference truth table prepared from simulations. Once the data obtained from the output of hardware implementation were successfully compared, resources and execution speed was compared against other implemented division processes to support the proposed algorithm's initial hypothesis. Table 3 represents the resource utilization of the divider circuit implemented based on the USP-Awadhoot division algorithm. The proposed division algorithm requires 238 LUT logic slices, 5 flip-flop slice registers, 140 latch slice registers, 8 seven input mux, and a total of 37 bounded I/o's for providing input operands and reading quotient and remainder outputs. The proposed division algorithm's working speed is given in terms of working frequency, equal to 100MHz with a power dissipation of 5.658 watts.

TABLE III
RESOURCE UTILIZATION OF PROPOSED DIVISION ALGORITHM

Sr. No.	Resource Parameter	Quantity
1	Slice LUT (Logic)	238
2	Slice LUT (Memory)	0
3	Slice Register (Flip-flop)	5
4	Slice Register (Latch)	140
5	MUX (F7)	8
6	MUX (F8)	0
7	DSP	0
8	Bounded IO	37
9	Power(W)	5.658
10	Frequency (MHz)	100

b. Discussion

Fig. 4 illustrates the comparative analysis of hardware resource utilization of the USP-Awadhoot division algorithm-based divider with other SRT-based radix – n algorithms-based divider. The proposed USP-Awadhoot division algorithm-based divider requires 238 slice logic LUT's and 145 flip-flop or latches, whereas radix-2 to radix-16 requires 1500 to 2100 slice logic LUT's and 1100 to 1200 flip-flop or latches [7].

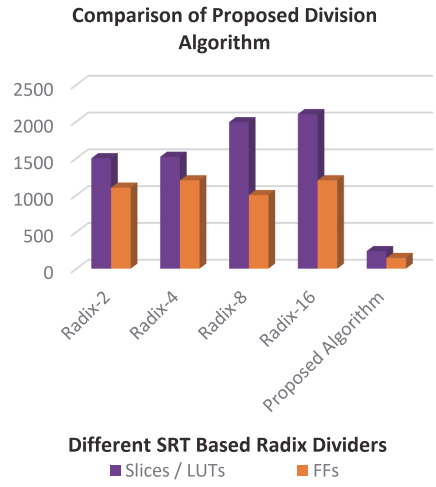


Fig.4. Comparative analysis of hardware resource utilization of the USP-Awadhoot division algorithm-based divider with other SRT based radix – n algorithms-based divider

Fig. 5 illustrates the comparative analysis of hardware resource utilization of the USP-Awadhoot division algorithm

with different functional iteration division algorithms. PST algorithm-based divider required 213 slice logic LUT's, 768 bytes of memory, and 28 DSP [27]. Fig. 6 illustrates the comparative analysis of the approximate conversion time of the USP-Awadhoot division algorithm-based divider with different digit recurrence algorithms-based divider. The proposed divider shows 250 ns for approximate conversion time, whereas Xilixs IP core required approximately 200 ns, SRT requires approximately 520 ns, restoring and non-restoring algorithms required approximately 320 ns and 160 ns [11]. Finally, we can summarize the performance of the proposed dividers as it indicates 65 % to 85 % area improvement over various dividers.

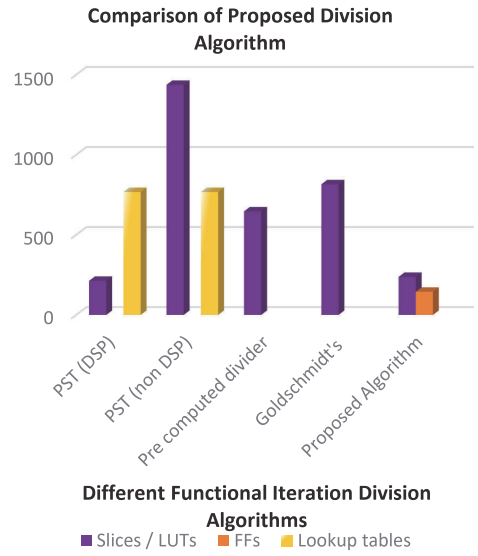


Fig.5. Comparative analysis of hardware resource utilization of the USP-Awadhoot division algorithm-based divider with different functional iteration division algorithms-based divider

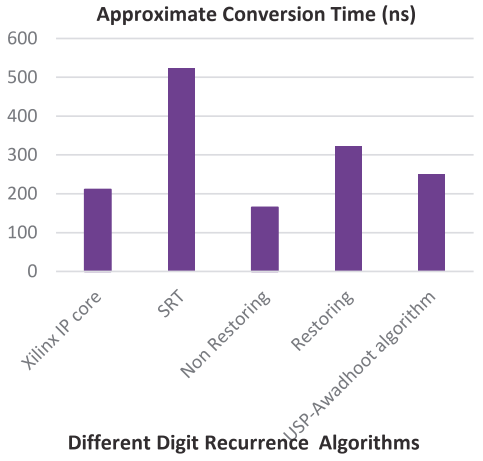


Fig.6. Comparative analysis of approximate conversion time of the USP-Awadhoot division algorithm-based divider with different digit recurrence algorithms-based divider

V. CONCLUSION

Digit recurrence division algorithms are easy to implement but critical to control quotient conversion logic, especially with SRT digit recurrence algorithm-based dividers. Based on the comparative analysis of experimental data of the proposed USP-Awadhoot algorithm-based divider, we conclude that the proposed USP-Awadhoot algorithm-based divider circuit is successfully designed and verified by implementing it using Vivado 2016, and Quartus prime lite simulation software and the Cyclone IV development board manufactured based on Altera EP4CE6E22C8N Cyclone IV FPGA and Zybo development board based on Xilinx Zynq XC7Z010. Even though the proposed USP-Awadhoot algorithm-based divider is implemented with only sequential and combinational hardware architecture, thus it requires more conversion time and works on moderate frequency up to 100 MHz. Also, from the resource utilization Table III and comparative analysis Fig. 4, 5, and 6, we can conclude that the proposed division algorithm shows remarkable improvement in the implementation area requirements compared to SRT base digit recurrence and functional iteration-based dividers. It suggests that the use of a proposed algorithm is suitable for an embedded system where the area is an important resource to maintain as low as possible.

VI. FUTURE WORK

The target for future works

- As the current implementation verifies the successful implementation of the proposed divider on different FPGAs, the next target is to design a dedicated integrated circuit IP. The first step is to design a physical layout, starting from the floor plan, which decides which circuit component is placed in which area and extracts the parasitic values to prepare the final layout for fabrication.
- Another future work target is to improve the working frequency and conversion time. To do so, we have to fuse some intermediate functional blocks like separate addition, and multiplication can be performed in fused mode like fused multiply-add (FMA). We need to test implementation and verify the resource utilization concerning the proposed divider to validate changes.
- Current implementation validated the successful implementation using combinational circuits. Thus, reducing the area and hardware resource utilization is also a future target. Some processes involved in the proposed divider can be represented as the different hardware architecture like pipelined architecture, parallel architecture, array structure, and cascade structure. Thus, it is required to validate the usage of different architectures and compare their resource utilization to prove the usability of the proposed divider in different working environments with different requirements. It gives detailed implementation results to choose the best suitable architecture implementation of the proposed divider in various applications as per their time, area, and power requirements.
- We have to verify the performance of the proposed divider in various applications like image processing,

particle detection, and signal processing. Complex number arithmetics is very important and critical, which requires careful design and more hardware resources. It is also very helpful in various essential engineering applications such as acoustics pulse reflectometry, astronomy, non-linear radio frequency measurements, control theory application such as finding out root locus, Nyquist plot, and microwave system frequency response. Our next target is to verify implementation resource utilization of the proposed divider for complex number computation.

ACKNOWLEDGMENT

This project has received funding from the Estonian Research Council Institutional Research Project PRG780, EAS - Enterprise Estonia under project number: EU60351 and partly from the European Union's Horizon 2020 Research and Innovation Program under Grant 668995. 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] R. K. L.Trummer, "A High-Performance Data-Dependent Hardware Integer Divider," master thesis, Institute of Computer Science and Systems Analysis, Paris Lodron University, Salzburg, May 2005.

[2] Donald G. Bailey, "Space Efficient Division on FPGAs," Electronics New Zealand Conference 2006, p-p – 206-211.

[3] Jyotika Kumari and Dr.M.Y.Yasin "Design and Soft Implementation of N-bit SRT Divider on FPGA through VHDL," International Journal for Innovations in Engineering, Science and Management, Volume 3, Issue 4, April 2015, ISSN 2347 – 7911, p.p- 13 – 19.

[4] Narendra K., S. Ahmed, S. Kumar, Asha G.H. "FPGA Implementation of Fixed-point Integer Divider Using Iterative Array Structure," International Journal of Engineering and Technical Research (IJETR) ISSN: 2321-0869, Volume-3, Issue-4, April 2015, p.p-170-179.

[5] E. Matthews, A. Lu, Z. Fang, and Lesley S., "Rethinking Integer Divider Design for FPGA-based Soft-Processors," 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), DOI: 10.1109/FCCM.2019.00046.

[6] K. Tatas, D. J. Soudris, D. Siomos, M. Dasygenis, and A. Thanailakis, "A Novel Division Algorithm For Parallel And Sequential processing," 9th International Conference on Electronics, Circuits, and Systems, Dubrovnik, Croatia, Croatia, 10 December 2002, ISBN: 0-7803-7596-3,p.p–553-556.

[7] K. D. TOCHER, "Techniques Of Multiplication And Division For Automatic Binary Computers," Quart. Journ. Mecta. and Appd. Math., Vol. XI, Pt. 3, 1958 p.p – 364-384.

[8] H. Asai, "A Recursive Radix Conversion Formula And Its Application To Multiplication And Division," Comp. and maths with appls., Vol. 2. pp. Z55-265 Pergamon Press 1976.

[9] Nikolay Sorokin, "Implementation of high-speed fixed-point dividers on FPGA," Journal of Computer Science & Technology; vol. 6, no. 1 ISSN: 1666-6038, April 2006, p.p- 8-11.

[10] Reza Z, Mehdi K, Arash F, Ali Kusha, Saeed S, Massoud P, "SEERAD: A High Speed yet Energy-Efficient Roundingbased Approximate Divider" Design, Automation & Test in Europe Conference & Exhibition – DATE 2016, ISSN: 978-3-9815370-7-9, p.p- 1481-1484.

[11] Elizabeth A, Suganthi V, and Seok-Bum Ko "Approximate Restoring Dividers Using Inexact Cells and Estimation From Partial Remainders" Ieee Transactions On Computers, Vol. 69, No. 4, April 2020, p.p- 468-474.

[12] L. Chen, J. Han, W. Liu, F. Lombardi, "Design of Approximate Unsigned Integer Non-restoring Divider for Inexact Computing" Great Lakes Symposium on VLSI, GLSVLSI- 2015, Pittsburgh Pennsylvania USA, ISBN- 978-1-4503-3474-7, May 2015, p.p- 51-56.

[13] N. Jamadagani, Jo Ebergen, "An Asynchronous Divider Implementation" IEEE 18th International Symposium on Asynchronous Circuits and Systems, 1522-8681/12, 2012 IEEE, p.p- 97-104.

[14] P. Saha, D. Kumar, P. Bhattacharyya, A. Dandapat, "Vedic division methodology for high-speed very large scale integration applications" Journal of Engineering; Accepted on 7 January 2014 Vol. 2014, Iss. 2, pp. 51–59.

[15] M. Reddy, Vasantha MH, N. Kumar, D. Dwivedi, "Design of Approximate Dividers for Error Tolerant Applications" IEEE 61st International Midwest Symposium on Circuits and Systems-MWSCAS ISBN- 978-1-5386-7392-8/18, 2018, p.p- 496 – 499.

[16] Andrew Kaplan, "Math on Call: A Mathematics Handbook," Published 2004.

[17] Bassarear, "Mathematics for Elementary School Teachers," Fourth Edition book, Publisher: Richard Stratton 2008.

[18] S. F. Oberman and M. J. Flynn "Division Algorithms and Implementations," Ieee Transactions On Computers, Vol. 46, No. 8, August 1997, p.p – 833 – 854.

[19] S. Dixit and Mohd. Nadeem, "FPGA Accomplishment of a 16-Bit Divider," Imperial Journal of Interdisciplinary Research (IJIR), Vol-3, Issue-2, 2017, ISSN: 2454-1362, p.p – 140 – 143.

[20] Muhd. Kasim, T. Adiono, Muhd. Fahreza, "FPGA Implementation of Fixed-Point Divider Using Pre-Computed Values," The 4th International Conference on Electrical Engineering and Informatics ICEEI 2013, Procedia Technology 11 (2013), p.p. - 206 – 211.

[21] G. Sutter, G. Bioul, J-P Deschamps, "Comparative Study of SRT-Dividers in FPGA," Becker J., Platzner M., Vernalde S. (eds) Field Programmable Logic and Application. FPL 2004. Lecture Notes in Computer Science, vol 3203. Springer, Berlin, Heidelberg, p.p – 209-220.

[22] R..S.Hongal, Anita D.J., "Comparative Study of Different Division Algorithms for Fixed and Floating-Point Arithmetic Unit for Embedded Applications," International Journal of Computer Sciences and Engineering, Volume-4, Issue-9, E-ISSN: 2347-2693, p.p- 48-54.

[23] Sukhmeet Kaur1, Suman2, Manpreet Singh Manna3, Rajeev Agarwal, "VHDL Implementation of Non-Restoring Division Algorithm Using High-Speed Adder/Subtractor," International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering, Vol. 2, Issue 7, July 2013, ISSN (Print): 2320 – 3765 ISSN (Online): 2278 – 8875, p.p – 3317 – 3324.

[24] N. Boullis, A. Tisserand, “On digit-recurrence division algorithms for self-timed circuits,” Inriainstitut National De Recherche En Informatique Et En Automatique, Research Report RR-4221, inria-00072398, July 2001, ISSN 0249-6399 ISRN INRIA/RR—4221.

[25] J. E. Robertson, "A New Class of Digital Division Methods," IRE transactions on electronic computers, Volume: EC-7, Issue: 3, Sept. 1958. P.p – 218 – 222.

[26] C. S. Wallace, "A suggestion for a fast multiplier," IEEE Trans. Electronic Computers, vol. EC-13, pp. 14-17, February 1964.

[27] J. Liu, M. Chang, and C.-K. Cheng, "An Iterative Division Algorithm for FPGAs," FPGA'06, Feb. 22–24, 2006, Monterey, California, USA, ACM 1595932925/06/0002, pp. 83-89.

Appendix 3

Publication III

Appeared in:
Patankar, Udayan; Koel, Ants; Patankar, Sunil; Flores, Miguel;
"Division Method and Circuit" in PTC the International Patent System, International Bureau of the World Intellectual Property Organization, application no.: PCT/IB2021/054942, submission no.: 054942, Date: 06 June 2021; published on- 15-12-2022, publication no- WO2022259009.
https://patentscope.wipo.int/search/en/detail.jsf?docId=WO2022259009&_cid=P20-LEVJIH-48100-1

PATENT COOPERATION TREATY

From the
INTERNATIONAL SEARCHING AUTHORITY

To:

see form PCT/ISA/220

PCT

WRITTEN OPINION OF THE INTERNATIONAL SEARCHING AUTHORITY

(PCT Rule 43*bis*.1)

Date of mailing
(day/month/year) see form PCT/ISA/210 (second sheet)

Applicant's or agent's file reference
see form PCT/ISA/220

FOR FURTHER ACTION
See paragraph 2 below

International application No.	International filing date *(day/month/year)*	Priority date *(day/month/year)*
PCT/IB2021/054942	06.06.2021	

International Patent Classification (IPC) or both national classification and IPC
INV. G06F7/535

Applicant
AWADHOOT LABS OÜ

1. This opinion contains indications relating to the following items:

- ☒ Box No. I Basis of the opinion
- ☐ Box No. II Priority
- ☐ Box No. III Non-establishment of opinion with regard to novelty, inventive step and industrial applicability
- ☐ Box No. IV Lack of unity of invention
- ☒ Box No. V Reasoned statement under Rule 43*bis*.1(a)(i) with regard to novelty, inventive step and industrial applicability; citations and explanations supporting such statement
- ☐ Box No. VI Certain documents cited
- ☒ Box No. VII Certain defects in the international application
- ☒ Box No. VIII Certain observations on the international application

2. **FURTHER ACTION**

If a demand for international preliminary examination is made, this opinion will usually be considered to be a written opinion of the International Preliminary Examining Authority ("IPEA") except that this does not apply where the applicant chooses an Authority other than this one to be the IPEA and the chosen IPEA has notifed the International Bureau under Rule 66.1*bis*(b) that written opinions of this International Searching Authority will not be so considered.

If this opinion is, as provided above, considered to be a written opinion of the IPEA, the applicant is invited to submit to the IPEA a written reply together, where appropriate, with amendments, before the expiration of 3 months from the date of mailing of Form PCT/ISA/220 or before the expiration of 22 months from the priority date, whichever expires later.

For further options, see Form PCT/ISA/220.

Name and mailing address of the ISA:	Date of completion of this opinion	Authorized Officer
European Patent Office D-80298 Munich Tel. +49 89 2399 - 0 Fax: +49 89 2399 - 4465	see form PCT/ISA/210	Prins, Leendert Telephone No. +49 89 2399-0

Box No. V Reasoned statement under Rule 43*bis*.1(a)(i) with regard to novelty, inventive step or industrial applicability; citations and explanations supporting such statement

1. Statement

Novelty (N)	Yes:	Claims	1-16
	No:	Claims	
Inventive step (IS)	Yes:	Claims	1-16
	No:	Claims	
Industrial applicability (IA)	Yes:	Claims	1-16
	No:	Claims	

2. Citations and explanations

see separate sheet

Box No. VII Certain defects in the international application

The following defects in the form or contents of the international application have been noted:

see separate sheet

Box No. VIII Certain observations on the international application

The following observations on the clarity of the claims, description, and drawings or on the question whether the claims are fully supported by the description, are made:

see separate sheet

Appendix 4

Publication IV

Appeared in:
Patankar, Udayan; Koel, Ants; Flores, Miguel;
"Novel Data Dependent Divider Circuit Block Implementation for Complex Division and Area Critical Applications," in NATURE Scientific Reports. Sci Rep 13, 3027 (2023). https://doi.org/10.1038/s41598-023-28343-3.

scientific reports

Check for updates

OPEN

Novel data dependent divider circuit block implementation for complex division and area critical applications

Udayan S. Patankar[1]✉, Miguel E. Flores[2] & Ants Koel[1]

This article elaborates on the state-of-the-art novel Udayan S. Patankar (USP)-Awadhoot algorithm for distinctive implementation area improvement for area-critical electronic applications. The proposed USP-Awadhoot divider is a digit recurrence class, but it can be flexibly implemented as a restoring or nonrestoring algorithm. The implementation example indicates the use of the Baudhayan-Pythagoras triplet method in association with the proposed USP-Awadhoot divider. The triplet method provides an easy way to generate Mat_Term1, Mat_Term2, and T_Term, which are further utilized with the proposed USP-Awadhoot divider. The USP-Awadhoot divider is implemented in three parts. First is preprocessing circuit stage for executing a dynamic separate scaling operation on input operands, ensuring the inputs are in the correct form. Second is the processing circuit stage for implementing the conversion logic expressed by the Awadhoot matrix, and third is the postprocessing circuit stage for recombining the individual results into the final result. The proposed divider works upto 285 MHz frequency with a power estimation of 3.366 W, also significantly improves the chip area requirements over those of the commercially and noncommercially implemented solutions.

Enhancement in the semiconductor manufacturing industry has proven valuable and innovative for existing applications such as communications, transport, signal processing, and computation, where mathematics plays a vital role and enables the evolution of new fields of work and study, mainly data protection, statistical data analysis, computational processing, signal processing, artificial intelligence, image processing, high-performance graphics rendering systems (such as graphic processing units (GPUs), complex systems on chips, central processing units, biomedical equipment, fuzzy control, and space engineering[1–15]. Performance evaluations of division operation implementations typically fall into the latency range of tens of clock cycles to hundreds of clock cycles[16–22]. Researchers have focused more on creating better adders and multipliers instead of developing dedicated algorithms for division operations to improve the divider circuit's implementation performance. Therefore, the prospect of improving or developing a new algorithm is plausible. This article elaborates on the state-of-the-art novel Udayan S. Patankar (USP)-Awadhoot algorithm to achieve distinctive implementation area improvement. Furthermore, the sections below describe a divider implementation based on the state-of-the-art novel USP-Awadhoot algorithm; a statistical analysis of its implementation resources; a comparative discussion with different dividers, complex division operations, and area-critical application followed by the conclusion and the future work directions.

Division circuit block taxonomy. A study presented in[23] indicated the performance dependency of a sophisticated system on a division circuit block implementation. It stated that the slightest improvement, such as a 1% improvement in a division circuit block, can increase the original system performance by up to 20%. The hierarchical distribution of various classes of division algorithms is expressed as a division algorithm taxonomy in Fig. 1 based on conversion logic, hardware architecture, performance, and execution type[6,16,24–30].

Digit recurrence is one of the most trusted, implemented, researched, and commercially utilized division algorithms among all divider implementation classes. Restoring algorithms and some nonrestoring algorithms implement simple conversion logic but requiring longer conversion time and large areas. Although the conversion logic is simple, it is not suited for high-frequency applications due to latency problems. Functional iterative

[1]Tallinn University of Technology, Tallinn, Estonia. [2]Electronics School, Don Bosco University, Soyapango, El Salvador. ✉email: udayan.patankar45@gmail.com

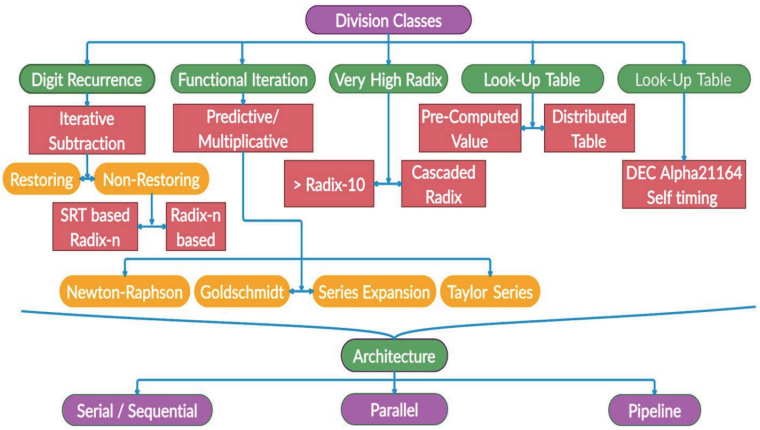


Figure 1. Division algorithm taxonomy.

class dividers compute a quotient bit based on the estimation or approximation of series expansion functions, such as the Newton-Rapson algorithm[31,32], Goldschmidt algorithm[11,33–36], Taylor series algorithm[11,37–40]. This approach utilizes multiplication instead of subtraction operations, reducing the number of required iterations, and can generate multiple quotient digits in one iteration with low latency, but the area required for a multiplier is higher than that of an adder or subtractor. This type of divider has a major drawback regarding quotient bit inaccuracy because of the direct rounding off of approximate solution values rather than infinitely precise values. The induced error depends on the accuracy of the initial estimation.

In the Newton–Raphson iteration method, which is limited to two multiplications and must proceed in series, a large error is generated due to the rounding off to approximate value. Reducing this error requires the introduction of a trade-off between the additional chip area required for the LUT and the latency of the divider. The Goldschmidt algorithm is another functional iterative divider whose major drawback is that it does not provide a remainder, making it useful only for floating-point division[36]. Another drawback is that 1's complement can prevent carry propagation delay, but it adds a new approximation error in each iteration. In Taylor series dividers, Taylor series expansion is used to calculate accurate anti-divisors (reciprocals) to reduce the error in the least-important bits of quotient precision with a parallel powering section that computes high-order terms, leading to extra hardware overhead and increased area requirements.

Variable-latency class[41,42] dividers are very rare due to their complexity and area constraints. A high radix divider[43] reduces the latency but requires a high-capacity LUT, which is impractical for implementation. The LUT class[27,44] requires storage such as read-only memory (ROM), which increases the area requirements of its implementations. For better performance, either optimized area and hardware resources are needed, or the latency cycles must be interrelated. Three types of hardware architectures can be used for divider implementations. A serial hardware architecture[25,45,46] requires higher latency and a larger conversion time, making it inappropriate for highly critical applications. A parallel hardware architecture[6,10,13] contrasts with serial architectures, requires multiple cores to work together simultaneously, makes synchronization critical, and has high area requirements, leading to increased implementation costs. A pipelined architecture[9,25,45,47] is the best choice for achieving parallelism in a sequential architecture with parallel processing. Some or all processes of division algorithms can be pipelined to achieve partial parallel processing. The radix based SRT division algorithm is one of the most implemented nonrestoring digit recurrence algorithms. The SRT algorithm named after Sweeney, Robertson, and Tocher is used in serial, parallel, pipelined, and cascaded architectures and various applications[16,25,28,48–64]. Although the SRT algorithm was the first choice for commercial implementations of the majority of soft and modern processors, such as Intel's Pentium processor[65], Xilinx's FPGA controllers[66], and the arithmetic logic units (ALUs) of complex hardware, it is restricted to specific low radix values (significantly less than 10). Radix-2 and radix-4 are the most implementable formats of the SRT algorithm. The main reasons for limiting the implementations of the SRT algorithm to low radix values are the increase in the quotient selection logic's criticality and the enormous increase in the area requirements of storing LUTs for this logic. It results in the failure to follow the execution cycle, as it requires multiple clock cycles for execution.

Complex divider. A software or hardware divider forms complex numbers based on the conventional formula using a complex conjugate, where z_1, and z_2 are two complex numbers consisting of real and imaginary parts. The divide-and-conquer concept must be used to implement two separate dividers for a complex number's real and imaginary parts. In the end, these two parts must be connected into one part to represent a complex quotient and the remainder as a final result. Many different approaches have recommended alternate number system for complex number representation as a single entity instead of separate real and imaginary parts, but it increases the complexity of conversion logic, resulting in area overhead[67–71]. A software or hardware divider forms complex numbers based on the conventional formula mentioned in (2), where z_1, and z_2 are two complex

numbers that may lead to overflow or underflow conditions when the operands are near the extreme ends of the representable range[72].

$$z_1 = x_1 + iy_1 \; and \; z_2 = x_2 + iy_2 \tag{1}$$

$$\frac{z_1}{z_2} = \frac{(x_1x_2 + y_1y_2)}{(x_2^2 + y_2^2)} + \frac{(x_2y_1 - x_1y_2)i}{(x_2^2 + y_2^2)} \tag{2}$$

Smith's algorithm solved this drawback by providing a more robust calculation, which was further enhanced by the Stewart process. Ultimately, the Stewart process makes the algorithm more complex rather than making it more robust. It also lacks a guarantee regarding the correctness of the rounding process for the quotient's real and imaginary parts during complex division. The hardware implementation of Smith's division algorithm is unsuitable because of the overhead caused by the costliest hardware components needed[72]. The hardware implementation of a single divider could be critical and in the complex divider, we must implement two sets of dividers for the real and imaginary parts of the given complex number, restricting the SRT divider to low radix to keep low area overhead and less critical conversion logic. In the case of a functional iteration divider, the correctness of the obtained result depends on the closeness of the reciprocal value selected in the initial iteration. Among several approaches for high-radix complex dividers, the best is a prescaled divider[72,73], where the divisor and dividend are multiplied by the same scaling factor so the resultant divisor must be close to unity. The main drawback of this method is that it requires an extra full-width divider for calculating the scaling factor.

Research questions. Many researchers have worked on various parameter improvement techniques, such as prescaling operands, carry-save remainders, array implementations, truncations, and differential LUTs, leading to the possibility of developing a new technique or combination of fast or moderate methods in terms of time and area efficiency. The current article is concerned with the following research problems/questions.

- Investigate the theory of conversion logic to develop a dynamic separate scaling operation/factor for input operands. Here separate scaling operations/factors mean one for the dividend and another for the divisor. Also, dynamic means different values for separate scaling operations/factors for different combinations of input operands.
- Improve the implementation area requirements to realize a dedicated divider circuit.
- Reduce the criticality of conversion logic by avoiding overlapping regions in quotient selection.

We propose a digit recurrence divider based on a state-of-the-art novel USP-Awadhoot algorithm for improving distinctive divider implementations with moderate operation speeds suitable for complex division and area-critical applications. In the following sections, we discuss the implementation of a Baudhayan-Pythagoras triplet method using a novel state-of-the-art USP-Awadhoot algorithm-based divider developed according to the ancient theories provided by Vedic mathematics during the early centuries. We also discuss the statistical analysis of implementation resources and elaborate on a comparative discussion with different dividers, followed by a conclusion and future work directions.

Complex division via the Baudhayan-Pythagoras triplet algorithm using a novel state-of-the-art USP-Awadhoot divider circuit block. In the present article, we discussed the unique way of complex division based on the Baudhayan-Pythagoras triplet method and the proposed novel state-of-the-art USP-Awadhoot divider circuit block. The use of the Baudhayan-Pythagoras triplet algorithm is possible because of the geometric properties of the complex numbers, which can be used to represent them via real and imaginary axis. The proposed complex division implementation is partitioned into three parts. The Baudhayan-Pythagoras triplet algorithm is used for the input circuit stage, ensuring the separation of the real and imaginary parts of complex number for further calculation. The second stage consists of a novel state-of-the-art USP-Awadhoot divider circuit block, which actually performs the division in real and imaginary parts of the complex number. The third stage consists of the recombination stage representing the final results in complex numbers. The Pythagorean theorem was known long before Pythagoras (570–500/490 BCE); Baudhayan (800–740 BCE) is said to be the pioneer of the Pythagorean theorem. Baudhayan formulated the relation between the hypotenuse and other sides of a triangle in terms of the area of the triangle in his book titled Baudhāyan Śulbasûtra, and in contrast, Pythagoras presented proof of the relationship between the hypotenuse and other sides of a triangle in terms of length[74,75] giving the equation

$$Baudhayan\text{-}Pythagoras \; Triplet(x, y, z) = T(x, y, z) \tag{3}$$

As $i = \sqrt{-1}$ or $i^2 = -1$, we can correlate the Baudhayan-Pythagoras triplet function $T(x, y, z)$ with the complex number, and we can represent a given complex number in terms of $T(x, y, z)$. The first two variables of the triplet are considered the real and imaginary coefficients of a given complex number. The following equations are used to develop the input circuit stage.

$$r_1 = x_1 + iy_1 \; and \; r_2 = x_2 + iy_2 \tag{4}$$

$$T(r_1) = f(x_1, y_1, z_1) \tag{5}$$

$$T(r_2) = f(x_2, y_2, z_2) \tag{6}$$

$$\frac{T(r_1)}{T(r_2)} = \frac{f(x_1, y_1, z_1)}{f(x_2, y_2, z_2)} = \left[\left(x_1x_2 + y_1y_2\right), \left(x_2y_1 - x_1y_2\right), z_2^2\right] \tag{7}$$

A detailed list of essential terms associated with the input circuit stage of complex divider implementation, as shown in Fig. 2, given below:

- Complex number one is termed $r_1 = x_1 + y_1 i$.
- Complex number two is termed $r_2 = x_2 + y_2 i$.
- The dividend of a complex number is termed "C_D_d".
- The divisor of a complex number is termed "C_D_r".
- The real number coefficient of the dividend of a complex number is termed "xD_d".
- The imaginary number coefficient of the dividend of a complex number is termed "yD_d".
- The real number coefficient of the divisor of a complex number is termed "xD_r".
- The imaginary number coefficient of the divisor of a complex number is termed "yD_r".
- The first triplet product term is named "TP_Term1".
- The second triplet product term is named "TP_Term2".
- The third triplet product term is named "TP_Term3".
- The fourth triplet product term is named "TP_Term4".
- The triplet term is named "T_Term".
- The first triplet matrix term is named "Mat_Term1".
- The second triplet matrix term is named "Mat_Term2".

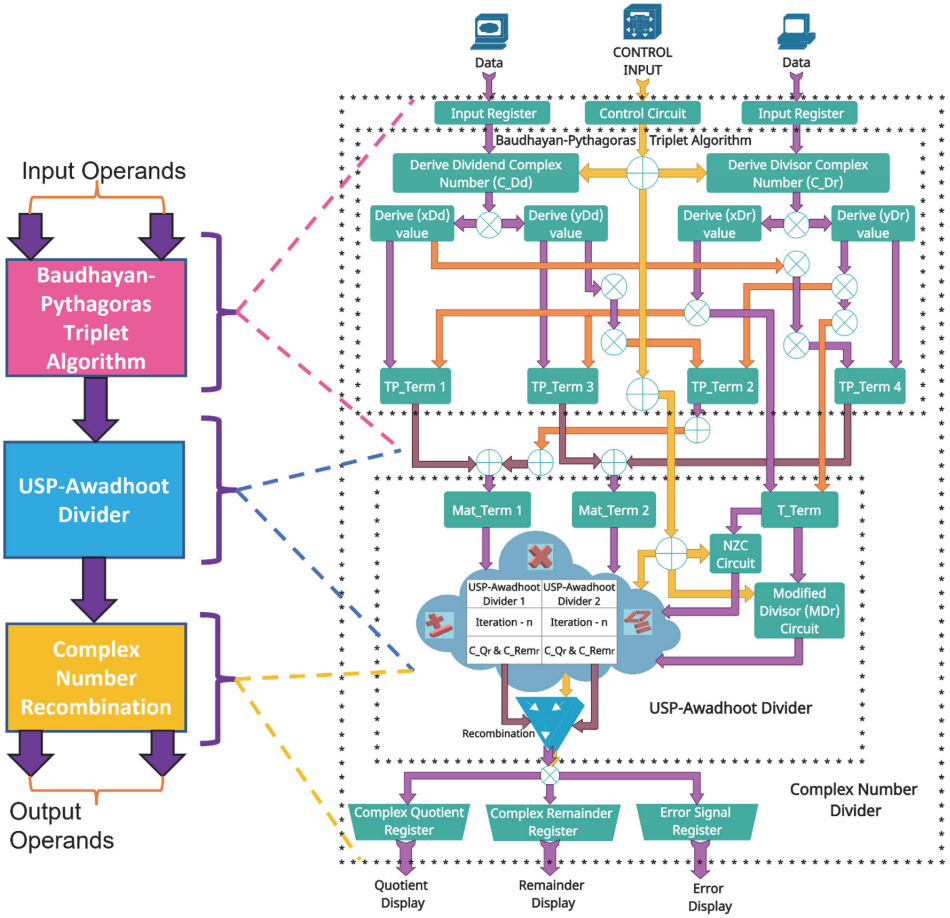


Figure 2. Schematic block diagram of the complex divider.

- The USP-Awadhoot dividend of a complex number is termed "C_D_{d1}".
- The USP-Awadhoot divisor of a complex number is termed "C_D_{r1}".
- The real number coefficient of the USP-Awadhoot quotient is termed "C_Q_r".
- The real number coefficient of the USP-Awadhoot remainder is termed "C_Rem_r".
- The imaginary number coefficient of the USP-Awadhoot quotient is termed "C_Q_i".
- The imaginary number coefficient of the USP-Awadhoot remainder is termed "C_Rem_i".
- The final quotient of a complex number is termed "C_Q".
- The final remainder of a complex number is termed "C_Rem".

Figure 2 illustrates the process of complex division implementation based on the Baudhayan-Pythagoras triplet algorithm and the proposed novel state-of-the-art USP-Awadhoot divider circuit block. Smith and Stewart's algorithm[72,73], represented by Eqs. (8) and (9), is generally used for software implementation of complex division, but the underflow and overflow conditions could occur during extreme distance between divisor and dividend, which results in the incorrect recombination of the real and imaginary part of the quotient and remainder[37]. We use the Baudhayan-Pythagoras triplet algorithm as an input circuit stage of the complex divider to eliminate this drawback.

$$\frac{r_1}{r_2} = \frac{x_1 + iy_1}{x_2 + iy_2} = \frac{\frac{y_2}{x_2}\left(x_1 + y_1\right)}{\frac{y_2}{x_2}\left(x_2 + y_2\right)} + i\frac{\frac{y_2}{x_2}\left(y_1 - x_1\right)}{\frac{y_2}{x_2}\left(x_2 + y_2\right)} \quad if \quad (x_2 \geq y_2) \tag{8}$$

$$\frac{r_1}{r_2} = \frac{x_1 + iy_1}{x_2 + iy_2} = \frac{\frac{x_2}{y_2}\left(x_1 + y_1\right)}{\frac{x_2}{y_2}\left(x_2 + y_2\right)} i\frac{\frac{x_2}{y_2}\left(x_1 - y_1\right)}{\frac{x_2}{y_2}\left(x_2 + y_2\right)} \quad if \quad (x_2 \leq y_2) \tag{9}$$

The first stage of the Baudhayan-Pythagoras triplet algorithm circuit block of the proposed divider separates the real and imaginary parts of the input operands and considers their real numeric values, especially the imaginary number's real numeric value without considering the imaginary unit (i), which is recombined at the final recombination circuit block. During the Baudhayan-Pythagoras triplet algorithm circuit block, all operands are processed to develop the Mat_Term1, Mat_Term2, and T_Term values/signals as an output of the first stage of the proposed divider. The logic behind Mat_Term1, Mat_Term2, and T_Term is explained in the next section with Fig. 3. In the second stage, the USP-Awadhoot divider circuit block receives Mat_Term1, Mat_Term2, and T_Term values/signals from the first stage. The Mat_Term1, Mat_Term2, and T_Term values/signals are essential to keep the operations under bounded conditions and avoid underflow and overflow conditions. During the second stage, the USP-Awadhoot divider circuit block generates two sets of the quotient and remainder values/signals separately as an output. In the final stage, the recombination circuit block rearranges quotients and remainders into complex numbers by adding an imaginary unit (i) with the real numeric value of the imaginary coefficient. It provides resultant quantities for display, storage, or further communication. The detailed work is explained in the next section.

Working theory of the Baudhayan-Pythagoras triplet algorithm. Figure 3a–c illustrates the schematic diagram of the proposed Baudhayan-Pythagoras triplet algorithm circuit block implementation. It consists of three circuit stages: input, intermediate, and output. The different signals used in the Baudhayan-Pythagoras triplet algorithm circuit block implementation are grouped into input operand, control, output, and indicator signal groups.

The signals r_1 and r_2 are the input operand data signals; Mat_Term1, Mat_Term2, and T_Term are output signals; Valid_O/P and Error are indicator signals; and cd_enable, CLK, and RST are considered control signals. the control group CLK signal provides the timing reference signal for computation execution. The reference clock signal's period value is dependent on the working frequency. When the CLK signal continues generating the reference signal and the control group signals (cd_enable and RST) both possess low logic values, then the operation of the proposed circuit is in an idle state. The values of the input operand, output, and indicator group signals are in a high-impedance tri-state condition during the idle state. As shown in Fig. 3a, the input operand signals r_1 *and* r_2 provide two complex numbers used to perform division operations based on the current statuses of the cd_enable, CLK, and RST control signals. In the input circuit stage, all real and imaginary parts of input operands are separated and stored in the input buffer and wait until the cd_enable signal is high (1) and reset (RST) is low (0), and the output circuit stage initializes the Mat_Term1, Mat_Term2, and T_Term signals from the output and indicator signal group to 00, assuring that the previous computation results are not involved in the current computation. Once the cd_enable signal is applied, this signal is used to develop a select signal and is stored in the control register to connect with further circuit stages. The input operand data is provided for further computation in the input circuit stage. The input operand data is stored into x_1, x_2, y_1 *and* y_2 buffers to extract $xD_d, yD_d, xD_r,$ *and* yD_r values, respectively, for the generation of signals B to G.

Figure 3b illustrates the intermediate circuit stage of the proposed Baudhayan-Pythagoras triplet algorithm implementation. The intermediate circuit stage receives signal B to signal G data from the input circuit stage and the O signal data from the output circuit stage. The forward signals B, C, D, F, H, and G are generated from the computation of TP_Term1 to TP_Term4, as

$$\text{TP\_Term1} = (xD_d \times xD_r) \tag{10}$$

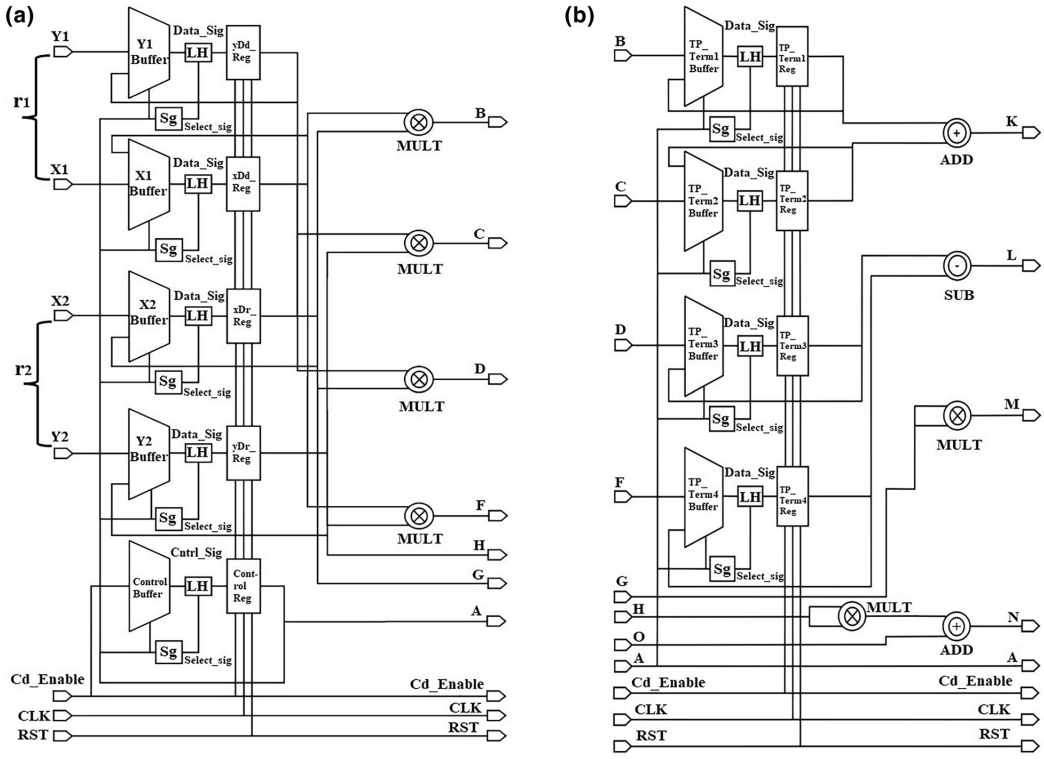


Figure 3. Schematic diagram of the proposed Baudhayan-Pythagoras triplet algorithm circuit block of the complex divider.

$$\text{TP\_Term2} = (yD_d \times yD_r) \tag{11}$$

$$\text{TP\_Term3} = (xD_r \times yD_d) \tag{12}$$

$$\text{TP\_Term4} = (xD_d \times yD_r) \tag{13}$$

The computed data are stored in separate buffers and provided for further computation based on the selected signal data to calculate partial Mat_Term1, partial Mat_Term2, and partial T_Term values. Signals K, L, M, and N, indicate the partial Mat_Term1, partial Mat_Term2, and partial T_Term values and transfer the respective data to the next circuit stage.

$$\text{T\_Term} = (xD_r)^2 + (yD_r)^2 \tag{14}$$

$$\text{Mat\_Term1} = \text{TP\_Term1} + \text{TP\_Term2} \tag{15}$$

$$\text{Mat\_Term2} = \text{TP\_Term3} - \text{TP\_Term4} \tag{16}$$

Figure 3c illustrates the output circuit stage of the proposed Baudhayan-Pythagoras triplet algorithm implementation. It receives signals K, L, M, and N from the intermediate circuit stage and stores the respective data in Mat_Term1, Mat_Term2, and T_Term buffers. The output circuit initializes the Valid_O/P and Error signals of the indicator group to 00 to ensure that no previously computed data are included. When the RST signal is deactivated, the cd_enable signal is activated during the initial state. Partial values are converted into the final Mat_Term1, Mat_Term2, and T_Term values based on the selected signal logic, and they are utilized as the Mat_Term1, Mat_Term2, and T_Term output group signals. These signals are connected with the proposed USP-Awadhoot divider circuit block and complex number recombination circuit to receive the final division result of complex input operands. The Valid_O/P and Error signal indicates computation completion and invalid working conditions, respectively. After completing the computation operation, depending on the completion of data computation, the Valid_O/P and Error are updated and validate the computation and O/P results, i.e.,

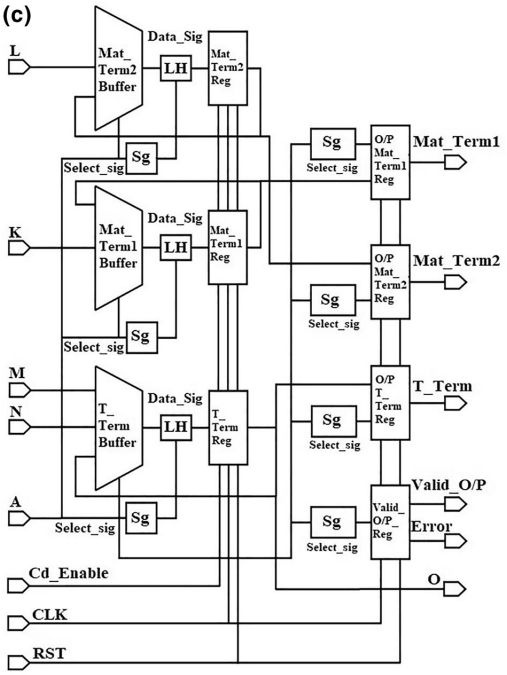


Figure 3. (continued)

whether the obtained values are correct or incorrect. If a high logic signal activates an RST signal, then the proposed divider circuit suspends its current computation operation state and resets it to the initialization state.

Working theory of the proposed novel state-of-the-art USP-Awadhoot divider circuit block. As discussed in the previous section, the novel state-of-the-art USP-Awadhoot Divider Circuit Block is the second major part of the proposed complex divider. The Baudhayan-Pythagoras triplet algorithm circuit is used as an input stage to arrange given operands in a required format to be supplied for the next stage of a complex divider to reduce the criticality of calculation and reduce the area overhead due to complex conversion logic. The major requirement of using novel state-of-the-art USP-Awadhoot divider circuit block is to reduce the implementation area for conversion logic. As we know, we have to use separate dividers for real and imaginary parts in maximum complex dividers. The SRT dividers are restricted to low radix as the implementation area increases with high radix, making conversion logic very complex due to overlapping regions. Functional iterative dividers take more area than SRT dividers, and sometimes the final results contain round-off errors. Thus, we proposed to use novel-state-of-the-art USP-Awadhoot Divider Circuit Block developed on the novel concept of a dynamic separate scaling operation/factor for input operands to reduce the implementation area for conversion logic and eliminate the overlapping region, which can simplify the conversion logic.

The working theory of the proposed novel state-of-the-art USP-Awadhoot divider circuit block is based on a three-stage algorithm developed and built on the ancient Indian mathematics (Vedic mathematics) rules. The detailed embodiments of the state-of-the-art USP-Awadhoot divider circuit block are presented herein with reference to the accompanying results, facts, and figures that describe a circuit implementation for achieving an area-effective implementation of the divider circuit with moderate time and power consumption. Figure 4 illustrates the functional block diagram of the proposed divider circuit block and is expressed in three circuit stages: Preprocessing circuit stage, Processing circuit stage, and Postprocessing circuit stage. Numbered blocks {101 to 104} stipulate preprocessing circuit stage components/elements as per the proposed algorithm. The preprocessing circuit stage accepts the input data (here, the dividend and divisor values) from the external channel, performing initial input processing and confirming that the data are in their correct form for the primary processing circuit stage. Arrows between the numbered blocks indicate data flow directions. Numbered blocks {105 to 107} stipulate processing circuit stage components/elements as per the proposed algorithm. This stage accepts the input data from the preprocessing circuit, performs iterations to implement the steps involved in the Awadhoot matrix, and provides separate group quotient bits that are further supplied to the postprocessing circuit stage. Numbered blocks {107 to 108} stipulate the postprocessing circuit stage components. This stage recombines separate quotient bits (hereafter termed group quotient bits) and presents the quotient and remainder data separately; output on the controlling signal verifies the correctness of the division operation performed by the circuit.

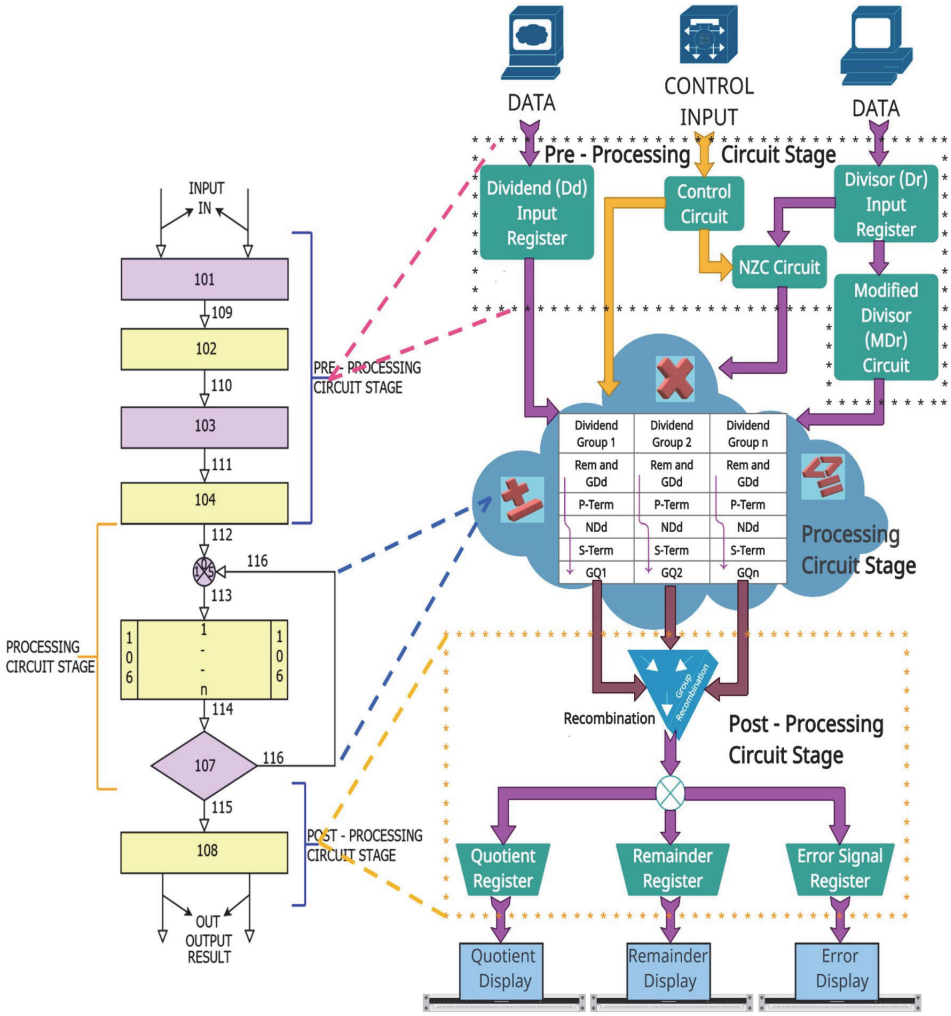


Figure 4. Schematic block diagram of the proposed USP-Awadhoot algorithm-based divider.

The first number is Divisor (D_r), and the second number, Dividend (D_d), are provided as input data signals for the proposed divider circuit, and Enable (E), Clock (CLK), Reset (RST), and Fd_Enable are provided as control input signals for the proposed divider circuit. The preprocessing circuit element covers the input data storage, control circuit, NZC circuit, and modified divisor circuit. The multiple outputs yielded by the preprocessing circuit stage are further fed to the processing circuit stage. The processing circuit stage iteratively constructs the core conversion logic, consisting of a dividend group, a p-term, an ND_d S-term and group quotients are arranged in a particular sequence of steps in one iterative circuit stage. The same element structure is used in all iterative circuit blocks of the processing circuit stage. At the end of the processing circuit stage, all individual dividend group quotients and the remainder are passed to the postprocessing stage. The individual dividend group quotients are recombined to form the final quotient and remainder in the postprocessing stage. In the end, confirmation of the completion of a successful conversion is validated by the present value of the valid output signal and the error signal in the postprocessing circuit stage. Detailed descriptions of the three stages are explained further, but it is essential to understand the vital terms or elements used in these three circuit stages of the USP-Awadhoot technique for the hardware and circuit implementation of a divider (or simply the USP-Awadhoot division technique). This approach utilizes a variable conversion time based on the dividend grouping technique. It has the following important terminology.

- $D_d = [d_1 d_2 \ldots \ldots \ldots \ldots d_k]$, where "$D_d$" represents dividends with a maximum size of "k" digits.

- $Q = [q_1 q_2 \ldots \ldots \ldots q_{k-1}]$, where "Q" represents a quotient with a maximum size of "$k-1$" digits. Except for the condition in which both operands are single digits, the maximum digit size is "k" digits.
- $D_r = [d_1 d_2 \ldots \ldots \ldots d_k]$, where "$D_r$" represents a divisor with a maximum size of "k" digits.
- $R = [r_1 r_2 \ldots \ldots \ldots r_{k-1}]$, where "R" represents a remainder with a maximum size of "$k-1$" digits.
- $ND_r = [ndr_1 ndr_2 \ldots \ldots \ldots \ldots ndr_m]$, where "$ND_r$" represents a "New Divisor" with a maximum size of "m" digits. The range is defined as "$k-1 \leq m \leq k+1$".
- $FD = [fd_1]$, where "FD" represents a "Flag Digit" with a maximum size of a single digit with a fixed range of [1,2……9].
- $MD_r = [md_1 md_2 \ldots \ldots \ldots md_p]$, where "$MD_r$" represents a "Modified Divisor" with a maximum size of "p" digits. The range is defined as "$p \leq k-1$".
- $NZC = [nzc_1 nzc_2 \ldots \ldots \ldots nzc_p]$, where "NZC" represents the "Number of Zeroes Cancelled" with a maximum size of "p" digits. The range is defined as "$p \leq k-1$".
- $ND_d = [ndd_1 ndd_2 \ldots \ldots \ldots ndd_k]$, where "$ND_d$" represents a "Net Dividend" with a maximum size of "k" digits.
- $G_r D_d = [gdd_1 gdd_2 \ldots \ldots \ldots gdd_k]$, where "$G_r D_d$" represents a "Gross Dividend" with a maximum size of "k" digits.

After providing all inputs, at numbered block 101, the preprocessing circuit stage obtains a divisor (D_r) and dividend (D_d) with maximum word sizes of "k" digits. The widths of the dividend and divisor determine the circuit hardware requirements. Prior to storing the input in the operand registers, i.e., the dividend register and divisor register, the divisor (D_r) and dividend (D_d) operands undergo an input normalization process that confirms that the input operands' widths are within the permissible limits and are in the required frame format. The use of a hex number system aids in keep this stage simple in terms of the implementation. This condition is not a restriction. In the SRT divider implementation, multiple number systems were used to reduce the criticality in the input circuitry. One of the best examples for expressing this involves binary-coded decimal (BCD) numbers, as explained in[76], where BCD numbers are used to implement the radix-10 SRT divider. The proposed algorithm is represented with a hexadecimal number system to provide a robust frame structure for electronic implementation. It is not restricted to hexadecimal number systems and can be used with other number systems, such as binary, decimal, and octal systems. The controlling signal circuit generates reference signals to control individual elements of the proposed divider's three circuit stages; then, the divisor (D_r) undergoes a check for invalid conditions, i.e., division by zero. This would indicate an error signal at numbered block 108 derived by signal 115 and redistribute further for display or transmission in the postprocessing circuit stage upon detecting the invalid condition. In the false case, the divisor (D_r) is passed by 109 to numbered block 102, where the circuit obtains a flag digit (FD) and a new divisor (ND_r); this step follows the basic concept of obtaining the FD and ND_r.

Later, at numbered block 103, the FD and ND_r are used to obtain the modified divisor (MD_r) and the number of zeros canceled (NZC); this step follows the basic concept of obtaining an MD_r and the NZC with respect to Fig. 4. Furthermore, the values of the FD and NZC are supplied to numbered block 104 by the 111 path. At this stage, dividend sectioning/regrouping is performed, and dividend groups are given out based on the NZC value provided by the previous step. Unlike the various SRT implementations that utilize operand prescaling or truncation[65,77], a fixed number of dividend sectioning or partitioning operations are performed to enhance the implementation; the proposed divider performs a cross combination of divisor prescaling and dividend sectioning or partitioning, giving us the upper hand to achieve area efficiency in the division implementation.

As discussed, the hardware requirements depend on the operand size; the maximum number of iterative circuit elements never exceeds the maximum operand size. This suggests that if the operand size is 8 bits, then a maximum of 8 iterative circuit stages is needed. Nevertheless, the number of iterative circuit stages used in a particular conversion depends on the value of the NZC. Similar to the variable-latency class algorithms, the dynamic nature of iterative circuit stages provides flexible conversion clock cycles for every dividend-divisor combination 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. Once the NZC value is determined, the circuit completes the preprocessing circuit stage and arranges the dividends MD_r and FD in separate dividend groups as per the arrangement shown in the Awadhoot matrix by sending data to numbered blocks 105–107. The proposed divider's processing circuit stage performs a computational process on the Awadhoot matrix (following Fig. 4) to obtain the group quotient's value and the remainder. Block number 106 shows the iterative circuit stages; as discussed earlier, the maximum number of iterative circuit stages is not greater than the operand width size. Numbered block 107 is the condition checker, which confirms that the computation ends in the iterative circuit stage. Once block 107 ensures the completion of the computation, the data are passed to the postprocessing circuit stage at block 108 of Fig. 4. Figure 4 represents the group recombination circuit followed by a distribution circuit for the separate visualization or transmission of the quotient (Q) and the remainder (R). After computing the Awadhoot matrix, the individual group quotients are recombined as per the relative weights and form a final quotient (Q). The final residue or remainder is obtained from the last iterative circuit stage, depending upon the conversion status.

- First: Net Dividend = 0. This shows that the dividend (D_d) is completely divisible by the divisor (D_r), where the remainder (R) = 0 and the quotient (Q) = the partial quotient (PQ_n) formed by concatenating the individual group quotients (GQ_n).
- Second: Net Dividend = Divisor (D_r). This shows that the dividend (D_d) is completely divisible by the divisor (D_r), where the remainder (R) = 0 and the quotient (Q) = the partial quotient (PQ_n) + 1.

- Third: Net Dividend (ND_d) > Divisor (D_r). The remainder (R) = R_{AQ} the value obtained during the calculation of the additional quotient (AQ), and the quotient (Q) = the partial quotient (PQ_n) + the additional quotient (AQ), where the AQ is derived by initializing the count to zero, subtracting the divisor (D_r) from the last iteration of the net dividend (ND_d), and incrementing the count by one until we obtain a sub-result that is zero or less than the divisor (D_r).
- Fourth: Net Dividend (ND_d) < Divisor (D_r). The remainder = the value of the last iteration of the ND_r, and the quotient (Q) = the partial quotient (PQ_n).

The last step of the proposed divider recombines the individual group quotient (GQ_n) in the postprocessing circuit stage. Upon completing the conversion process, the final quotient and remainder are available, along with an error signal indicating the conversion's correctness and the presented data.

$$D_d = [d_1 d_2 \ldots \ldots \ldots \ldots d_k] \quad and \quad D_r = [d_1 d_2 \ldots \ldots \ldots d_k] \tag{17}$$

$$Division(Q, R) = f(D_d, D_r) = f(Awadhoot \quad matrix, Condition) \tag{18}$$

$$Awadhoot \quad matrix(GQ_n, R_n) = f(GD_d, MD_r, FD) \tag{19}$$

$$f(GD_d, MD_r, FD) = \sum_{n=1}^{k} [(R_{n-1}|GD_{dn}) + (P - Term)_n - (S - Term)_n] \tag{20}$$

$$Condition(Q, R) = f(ND_{dn}, PQ, AQ) \tag{21}$$

Therefore, after the last iteration of the Awadhoot matrix, based on the condition function, the final quotient and remainder value are calculated and represented as follows:

$$Division(Q, R) = (PQ, 0) \quad if \quad ND_{dn} = 0 \tag{22}$$

$$Division(Q, R) = [(PQ + 1), 0] \quad if \quad ND_{dn} = D_r \tag{23}$$

$$Division(Q, R) = (PQ, ND_{dn}) \quad if \quad ND_{dn} < D_r \tag{24}$$

$$Division(Q, R) = [(PQ + AQ), R_{AQ}] \quad if \quad ND_{dn} > D_r \tag{25}$$

where PQ is the partial quotient, AQ is the additional quotient and R_{AQ} is the remainder generated during the calculation of the additional quotient.

In most of the performance enhancement schemes utilized in divider circuit block implementations, scaling down the operands with a common static scaling factor is considered a preliminary option. Different methods may be available for calculating the scaling factor, but the same factor scales down both operands (divisor and dividend). Thus, even after scaling down, the relationship between the divisor and dividend remains the same. Considering that the dividend is (x), the divisor is (y), and both operands are scaled down by a common scaling factor (m), the relation between the dividend and divisor is expressed as

$$(x \to y) = (x_m \to y_m) \tag{26}$$

Example: If the dividend = 500, the divisor = 50, and they are scaled down by common factor 5, then

$$\left(\frac{500}{50} \times 100\right) = 1000\% \tag{27}$$

where the relationship between the scaled-down values is presented as

$$\left(\frac{100}{10} \times 100\right) = 1000\% \tag{28}$$

Several ways of finalizing the scaling factor may be available, but the same scaling factor is used to scale down both operands. By performing scaling, we can reduce the values of operands, which can reduce the number of iterations required to calculate the quotient bits; however, this does not allow us to reduce the divisor quantity beyond the preliminary relation. Even though it is possible to further scale down the dividend or divisor, it is not executed because of the divisor has reached its limits. Nevertheless, doing so increases the area overhead. As explained in Fig. 4, the preprocessing circuit consists of the control circuit, NZC circuit, and modified divisor circuit, confirming the use of a dynamic separate scaling operation/factor, reducing the number of iterations required for the quotient calculation and ensuring that different factors scale down the divisor and dividend. Partitioning the original dividend value into several group dividends is also considered to ease the process of designing the quotient bit selection logic. The preprocessing circuit provides a key improvement to achieve performance enhancement. The processing circuit stage provides restoring and nonrestoring functionality for executing the division steps described in the Awadhoot matrix. It also provides a second improvement in the form

of clearer and simpler quotient selection logic without overlapping conditions, unlike in the SRT divider, where the overlapping region is critical and causes severe losses in the case of misalignment of the overlapping area.

The Awadhoot matrix circuit elements of the processing circuit stage. Figure 5 illustrates the particular arrangement of the processing circuit stage elements of the proposed algorithm. The structure is termed the Awadhoot matrix. The Awadhoot matrix provides a computational arrangement of various aspects of the processing circuit stage of a proposed divider. Figure 5 shows that each column represents an individual iterative circuit stage, and each row represents the elements of the corresponding iterative circuit stage. We can use a single set or multiple sets of iterative circuit elements depending on which hardware architecture is considered for implementation.

The Awadhoot matrix arrangement is composed of the previous remainder (R_{n-1}), the group dividend (GD_{dn}), the group quotient of the previous iteration (GQ_{n-1}), the gross dividend (G_rD_{dn}), the flag digit (FD), the modified divisor (MD_r), the net dividend (ND_{dn}), the present quotient (Q_n) and the present remainder (R_n). The proposed divider circuit hardware requirement depends on the number of dividend groups made in the preprocessing stage of the divider, and the maximum possible number of dividend groups is related to the maximum width among the available operands. This Awadhoot matrix arrangement provides a detailed structure of the processing circuit, which can be realized by serial, parallel, or pipeline hardware architectures. In the present article, we compare the sequential combinational circuit implementation of the Awadhoot matrix. Under the idle condition, the values of the previous remainder (R_{n-1}) and the previous iteration group quotient (Q_{n-1}) are considered zero to avoid any computational errors in an iterative circuit. Upon the execution value of the first iteration circuit stage, the group quotient and remainder are used as the previous iteration group quotient (Q_{n-1}) and the previous remainder (R_{n-1}), respectively, for the next iteration circuit stage and depend on the number of dividend groups.

A detailed description of the Awadhoot matrix is given in the patent application. In short, a gross dividend (G_rD_{dn}) is derived from the previous remainder (R_{n-1}) and the present value of the group dividend at the first level of the iterative circuit stage of the Awadhoot matrix. Further simple addition and multiplication operations are performed with the gross dividend (G_rD_{dn}), previous iteration group quotient (GQ_{n-1}) and flag digit (FD) to derive the value of the net dividend (ND_{dn}), which is indicated by the p-term terminology in the Awadhoot matrix. Additional multiplication operations are performed depending on the condition of the present net dividend (ND_{dn}) value in comparison with the value of the modified divisor (MD_r) to obtain the value of the s-term. Depending on the comparison, the final value of the group quotient (GQ) and the present remainder (R_n) are calculated and presented for the next iterative circuit stage or postprocessing circuit stage. During the execution of the postprocessing circuit stage, all individual group quotient values are recombined together with the associative weights to form the final quotient value. Later, this final quotient and remainder value are displayed or transmitted to other circuits if necessary. We consider the hexadecimal number system to implement the proposed system due to its ease of use in digital systems and computer applications. Hexadecimal numbers

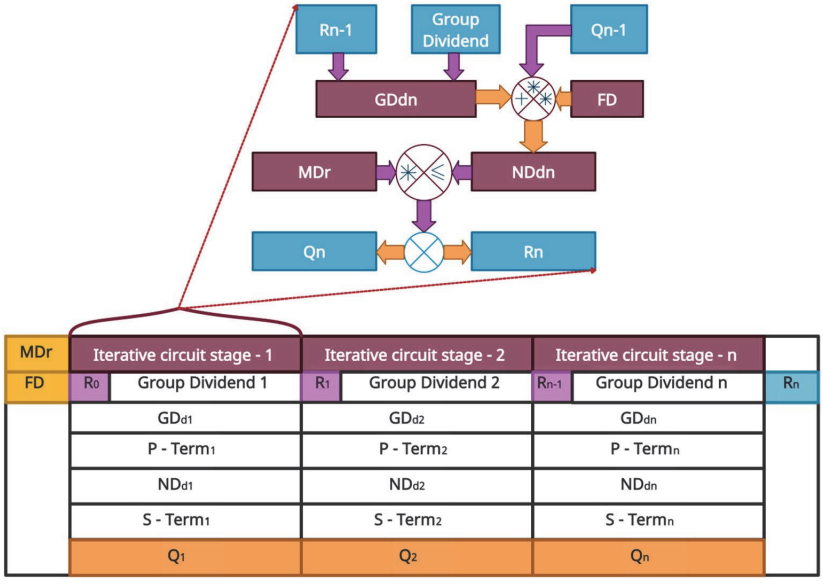


Figure 5. The Awadhoot matrix.

in digital electronics result in better readability and provide a frame structure with a fixed bit size to represent each decimal number in a digital form.

We obtain several representation forms in the binary representations of decimal numbers or digits. Sometimes they can be represented by one-bit equivalent binary numbers or multiple-bit binary numbers, whereas in the case of hexadecimal representation, every hexadecimal digit is defined as a frame of four binary bits. The fixed frame used for representing hexadecimal numbers in digital or binary form provides better support for performing operations such as shifting, comparing, and giving a simple logic for quotient bit selection in the processing circuit stage.

The use of a hexadecimal system also simplifies internal operations, such as concatenating digits in digital computations. A binary system could also be used, but hexadecimal numbers also provide the advantage of working with four bits per digit each time. In the binary system, the minimal number of bits to be considered for computation is one; in the case of a hexadecimal system, four binary bits are used, providing more clarity for understanding the computation process performed on a digital system with long bitstream data. The conversion of any digit value of any number system into a single digit by repetitively adding all digits is called a beejank or digital root. The beejank operation (digital root), alternate beejank operation, deviation, and alternate representation of addition and subtraction, which we perform during the iteration of the processing circuit stage, exhibit great ease of use in the quotient bit selection logic developed with the hexadecimal system.

The beejank approach is used to verify answers except for the binary number system; hence, we consider a hexadecimal number system in the quotient bit selection logic to confirm the correct selection of the quotient bit in a particular iteration. The same operations are performed with a number and its beejank; if both results are found to be the same, the answer is verified. The beejank does not indicate a deficiency in the minimum (zero) and maximum numbers. If the placement(s) of a digit (digits) in a number is/are interchanged, then this change is not indicated by the beejank. If the beejank is negative, then '9' is added to convert it to a positive value.

$$(F89A0BCD)_{16} = (F + 8 + 9 + A + 0 + B + C + D)_{16} \tag{29}$$

$$(F89A0BCD)_{16} = (4E)_{16} = (4 + E)_{16} = (12)_{16} = (3)_{16} \tag{30}$$

The difference between a number and its nearest base is called the deviation.

$$Number = (100F)_{16} \;\; Base = 1000 \;\; Deviation = 00F \tag{31}$$

During computation, the deviation also supports the representation of the addition or subtraction of two numbers as one hexadecimal number, which can reduce the number of steps required in the quotient bit selection logic and helps to reduce the area requirement and complexity of the quotient bit selection logic.

Implementation statistics and performance results analysis. The very high-speed integrated circuit (VHDL) hardware description language is used to develop the implementation idea based on the functional block diagram of the proposed USP-Awadhoot algorithm-based divider. To realize the theoretical concept and idea of the proposed state-of-the-art novel USP-Awadhoot algorithm-based divider, we develop a synthesizable architecture. This synthesizable architecture implementation provides a unified way of comparing and testing the proposed divider. To develop and implement the proposed USP-Awadhoot algorithm-based divider, we use the Vivado 2016 simulation tool with the Zybo development board based on Xilinx Zynq XC7Z010, XCZU7EV-FFVC1156-2-E with Zynq UltraScale + MPSoC and the Quartus Prime Lite simulation software with the Cyclone IV development board based on the EP4CE6E22C8N Cyclone IV FPGA manufactured by Altera to generate a truth table and cross-verify the simulation results by comparing the outputs separately. Here two different FPGA's (Xilinx and Altera) were used to test the correctness of the logical results when implemented with different structured FPGAs. Xilinx implementation and simulation statistics of the proposed divider considered further for comparison. The overall performance of the divider depends on the proposed algorithm for the data-dependent divider, which determines the latency for a particular input operand combination. To gain complete control over each implementation detail and make the synthesis as technology independent as possible, we create a set of components based exclusively on register transfer level (RTL) descriptions. In other words, each component is described by some structure composed of basic gates, and their connectivity is similar to that of the various implementations studied during the review.

We verify the simulation output of the proposed divider, which depends on the random number generator (RNG) outputs for the random and sequential input operand combinations covering complete bit or digit range operands, as proposed by the truth table. To verify whether the generated output is valid, we prepare a truth table for every possible operand combination, including the true or theoretical results. We execute this process for every possible input operand combination and compare its results with the truth table. Any differences in the results indicate the incorrectness of the calculation and are rectified via corrective actions. Also, all the possible input operand conditions are sequentially and randomly checked on both experimental test boards. We created one logic test bench board, as shown in Fig. 6, that can provide operand values with multiple word sizes. The logic test bench board is designed so that it can check minuscule bit size combinations and offer the flexibility to add extra bits to the input operands in cases with extended bit sizes. We tested the operation of the proposed USP-Awadhoot algorithm-based divider with static inputs supplied via separate single-pole single-throw rocker switches and a continuous sequence generator. We executed simulations with different clock frequencies and also tested by implementing them in both development boards to determine the working latency and conversion speed.

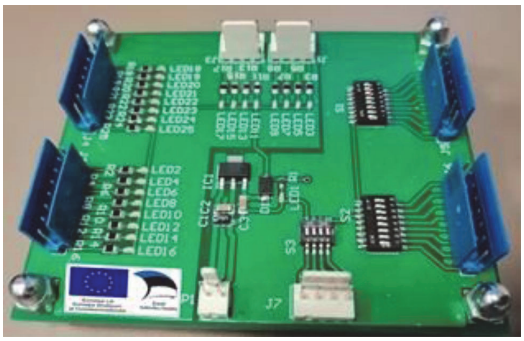


Figure 6. Logic test bench board.

After verifying the simulations and hardware implementations of different versions of the proposed divider circuit, the implementation statistics of every version are mapped to the LUTs, flip-flops, registers, multiplexers, latches, adder, subtractor, multiplier, basic gates, clock frequency, and power. In short, the number of transistors or gates used indicates the implemented area, hardware resource utilization, working frequency, and power consumption needed to realize the proposed USP-Awadhoot algorithm-based divider. Thus, the behavior of the proposed USP-Awadhoot algorithm-based divider is mapped to the latency performance function designed to keep track of the required clock cycles for a given pair of input operands. It is the best way to compare implementation statistics based on area, and the other approach for performance comparison is based on the latency time.

The physical implementation of the proposed divider based on the USP-Awadhoot division algorithm is performed with various hardware implementation techniques, referred to as the implementation versions. A study of different implementation versions indicates the total trade-offs between area, time, and power, so depending on the application, one must decide which implementation version should be utilized. Implementation versions V8.1 to V8.9 are based on 8-bit operands, whereas V16.1, V24.1 and V31.1 are the 16-bit operand, 24-bit operand, and 31-bit operand implementations, respectively. The variances in the resulting parameters for different versions of the proposed divider circuit indicate the effectiveness of the corresponding versions.

As mentioned above, version V8.1 is the first implementation version of the proposed USP-Awadhoot divider with 8-bit operands. All the steps involved in the preprocessing, processing, and postprocessing circuit stages are implemented sequentially, and the execution of each step is concluded in a separate clock cycle. However, it requires less implementation area (explained in a further section), resulting in the slowest implementation of the proposed divider circuit because storing the intermediate values generated during execution is not needed, as the next step is executed only after the completion of the first step. Version V8.2 is the successor to version V8.1; it includes storing intermediate values generated during the execution process, causing the area overhead to increase compared to that of the predecessor version (V8.1). Version V8.3 is the third version of the 8-bit implementation of the proposed USP-Awadhoot divider in the series utilizing separate clock cycle executions for each step of the preprocessing, processing, and postprocessing circuit stages. While implementing these versions, we change the ND_d, the remainder and the error signal calculation circuit and provide an extra buffer to hold the values. This improves the accuracy of the output by providing the correct remainder value. Version V8.4 initiates the concurrent execution concept to implement the proposed USP-Awadhoot divider. In this version, the circuit allows the data available at the input data lines to be stored at input registers, the most significant bits (MSB) are stored as separate hexadecimal integers, and the least significant bits (LSB) are stored as additional hexadecimal integers in an array of hexadecimal integer elements for the dividend. The same process is applied to the divisor. Concurrently, the preprocessing circuit stage formulates the FD and ND_r values by working only on the least significant hexadecimal part of the divisor.

V8.5, V8.6, V8.7, V8.8 and V8.9 are the successors of the V8.4 version of the proposed USP-Awadhoot divider implementation. In the V8.5 version of the implementation, the processing circuit stage concurrently executes the ND_d and GQ_n calculation steps. In the V8.6 version of the implementation, the processing circuit stage utilizes the predefined values for error conditions during the concurrent execution of the ND_d and GQ_n calculations. It reduces the area overhead and power consumption while increasing the execution speed. In the V8.7 version of the implementation, the processing circuit stage concurrently executes the residue/remainder and additional quotient calculations. Based on the concurrent execution process, the ND_d values are compared, and the condition selection circuit is activated in the postprocessing circuit stage to compute the final values for the quotient and residue as per the display requirements. In the V8.8 version of the implementation, the processing circuit stage introduces an extra buffer and counter in addition to those in the V8.7 version to improve the expected group residue/remainder calculations. This version improves the working clock cycle requirements and maintains the same area requirements as the V8.7 version of the proposed USP-Awadhoot divider implementation. In the V8.9 version of the implementation, the processing circuit stage introduces different logic to implement alternate conditions to the residue/remainder and additional quotient calculations. This version improves the implementation area or resource utilization by keeping same clock cycle requirements as the V8.8 version of the proposed USP-Awadhoot divider implementation. Versions V16.1, V24.1 and V31.1 are the 16-bit operand,

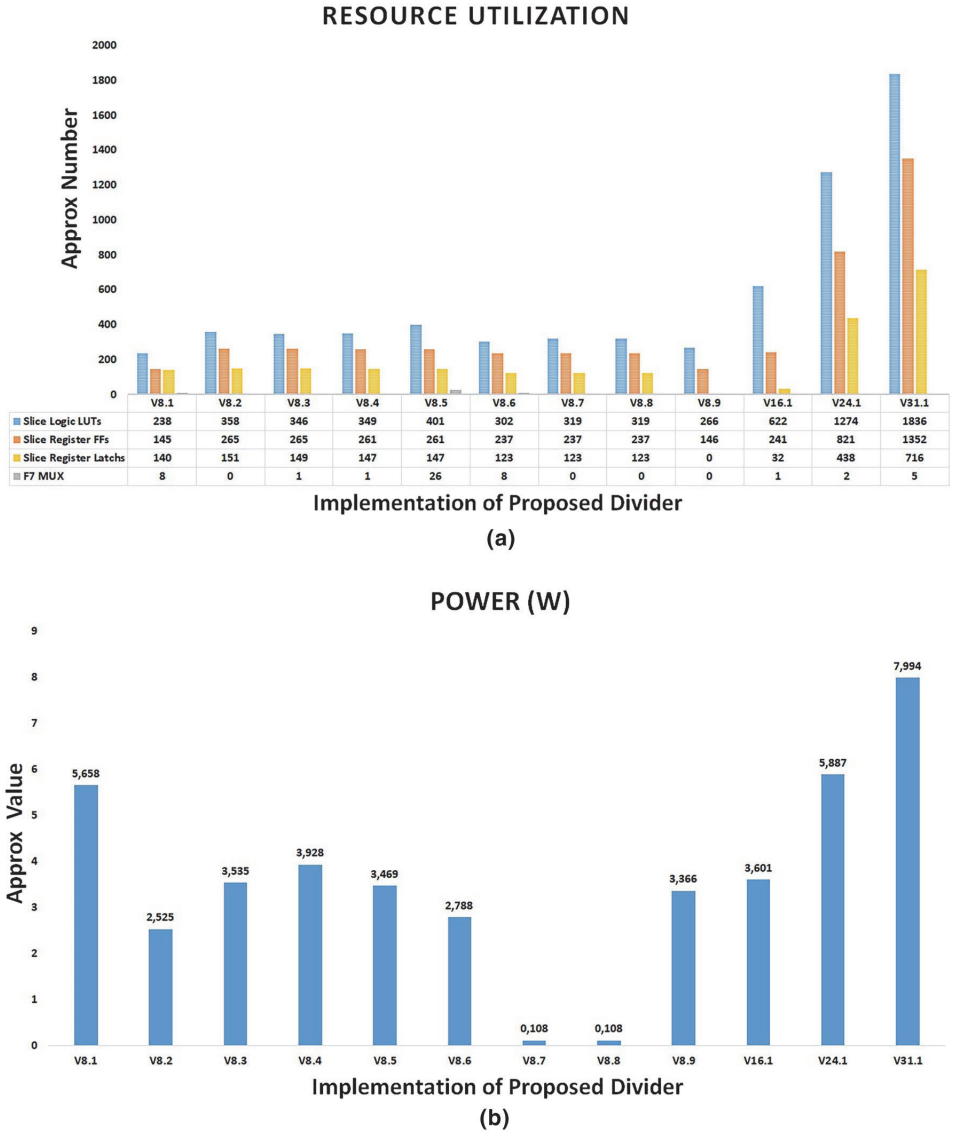


Figure 7. Hardware resource utilization of the proposed USP-Awadhoot algorithm-based divider.

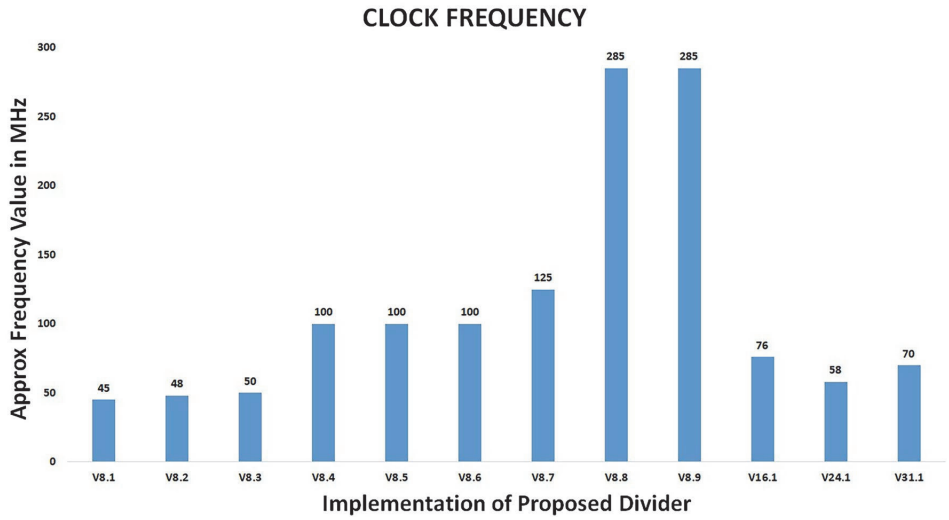


Figure 7. (continued)

24-bit operand, and 31-bit operand implementations, respectively, based on the modifications performed in the V8.9 version. V31.1 implementation uses 31 bit operands to avoid overflow condition during conversion.

Figure 7a–c shows hardware resource utilizations required by multiple versions of the proposed USP-Awadhoot algorithm-based divider implementations. The results shown in Fig. 7 are the actual data for the proposed circuit implementation based on Xilinx FPGA and Vivado 2016 simulation tool, which is considered as a baseline to compare with several other implementations to draw a comparative analysis. The circuit arrangements are different in terms of how they execute the various states of the logic flow state diagram of the proposed divider to improve the implementation area and enhance the operational performance with respect to the space, latency time, and power of the proposed divider circuit based on the USP-Awadhoot division algorithm. While implementing these versions, we must consider that area is an essential point when working with embedded systems. Based on the slice logic LUT graph of the hardware resource utilization, we confirm that each implementation requires a minimum of 238 counts of slice logic LUTs after comparing the different 8-bit implementation versions.

Slice logic represent the group of hardware resources necessary to create a configurable logic block. Every slice logic contains a fixed numbers of LUTs and slice register flip-flops; sometimes, they are accompanied by slice register latches and multiplexers. A LUT is a collection of logic gates that are hard-wired on an FPGA. LUTs store a predefined list of outputs for every combination of inputs and provide a fast way to retrieve a logic operation's output. A flip-flop is a circuit that is capable of two stable states and represents a single bit. A multiplexer, also known as a mux, is a circuit that selects between two or more inputs and outputs the selected input. Different FPGA families implement slices and LUTs differently. For example, a logic slice on a Virtex-II FPGA has two LUTs and two flip-flops, but a logic slice on a Virtex-5 FPGA has four LUTs and four flip-flops. Additionally, the number of inputs to a LUT, commonly two to six, depends on the selected FPGA family. A register is a group of flip-flops that stores a bit pattern. A register on an FPGA has a clock, input data, output data, and enabled signal ports. Every clock cycle, the input data are latched and stored internally, and the output data are updated to match the internally stored data.

While implementing the proposed divider, an approximately one hundred and forty-six slice register flip-flops, and 37 bounded input outputs are required in the case of an 8-bit implementation. The bounded I/O is divided into two groups of data operands, which are input, and output data lines and control lines used to control the divider's operation and indicate an error if it occurs during computation. Some implementation versions require additional seven-input multiplexers or eight-input multiplexers. Versions V8.1 and V8.6 require eight seven-input multiplexers, V8.3 and V8.4 require one seven-input multiplexer, and V8.5 requires twenty-six seven-input multiplexers. Along with the area and power analyses, frequency or cycle time calculations also form an important part of divider circuit analysis.

The proposed divider's implementation uses variable power (approximately from a minimum of 0.108 watts to a maximum of 7.9941 W). It works at up to 285 MHz depending on the implementation version selected, i.e., the 8-bit, 16-bit, 24-bit, or 31-bit implementation. Depending on the best suitable hardware resource utilization combination and a better performance analysis of the proposed USP-Awadhoot algorithm-based divider implementation, version V8.9 is the most suitable option for implementation in an application with a lower-to-moderate range of resource requirements. This version requires 266 slice logic LUTs, 146 slice register flip-flops, zero latches and 37 bounded input–outputs with no seven-input multiplexers, DSP, or eight-input multiplexers. Simulation confirms it works at a moderate frequency up to 285 MHz and requires 3.366 watts estimated power to run. For all the comparative studies we consider version V8.9 of the proposed divider circuit.

During each clock cycle, the processing unit can perform basic operations such as fetching instructions or data, accessing memory, and reading or writing data. The processing unit often requires multiple clock cycles to complete a single process. The processing unit's frequency is calculated depending on the clock cycles, also termed the cycles per second or frequency. Here, the clock frequency is considered the system frequency or the divider's working frequency. The clock frequency is also considered a reference point for executing different instructions during the implementation of a particular operation. The frequency of a processing unit is also known as the processor's clock speed. Clock speed is essential for determining the processor's overall performance. Since processors have different instruction sets, they may differ in the number of cycles needed to complete each instruction (or cycles per instruction (CPI)). Some processors can perform faster than others, even at slower clock speeds. This indicates that studying the clock cycles required for a particular conversion is essential.

Latency analysis is conducted in terms of clock cycles as the behaviour of the proposed USP-Awadhoot algorithm-based divider is mapped with a latency time performance function designed to keep a record of the number of clock cycles required given a pair of input operands. Simulations obtain the minimum and maximum performance of the proposed divider with analytically determining best and worst cases. The use of a latency time calculation in terms of clock cycles for determining the best and worst cases is rational when comparing divider performance. To consider the latency time performance of the proposed divider, we choose two options; the first is a sequential truth table that assures that each combination of input operands is considered during the execution, and its related data are stored.

The second option is RNG, is suitable for evaluating operands possessing larger word sizes with more significance. In the case of the data-dependent divider, the execution time depends not on how large the dividend is but on how far the divisor is from the dividend. The larger the distance between the dividend and divisor, the more execution time is needed. We conduct a clock performance analysis of the comprehensive range of dividends and divisors for 8-bit operands with the RNG and the sequential truth table, where the number of possible combinations is 65K; for higher bit sizes, only the RNG method is considered due to the possibility of billions of combinations. There are two possible ways to achieve variable-latency time in the division operation: one is by varying the frequency for performing the iteration process, and the second is by varying the number of iterations. One can provide a variable conversion rate or time by having variable latency. Latency is defined as the total time taken by an operation to generate the first output after providing an input; in other words, it is the total number of clock cycles required after providing inputs to develop the first result. We conduct a divider circuit clock performance analysis to understand the nature of the proposed divider.

The proposed division circuit based on the USP-Awadhoot algorithm executes all possible input operand combinations while performing the divider circuit's clock performance analysis. Every possible combination of dividend and divisor values is executed, giving us the details regarding the number of clock cycles required to compute a dividend–divisor combination. In the present article, we provide the implementation data for the eight-bit operands, suggesting that the divisor's width and the dividend's width are eight bits. The eight bits of each operand are expressed as two hexadecimal numbers, making it easier to execute the computation. We perform 65K combinations of input operands.

To represent the data in a standard format, we divide the dividend range into three sections with a low range of dividend values in the first region named dividend range 00, suggesting that the dividend has half-filled four-bit values or one-digit-lower hexadecimal values; a middle range of dividend values in the second region named dividend range 80, suggesting a range of half-filled four-bit values to six-bit values or lower two-digit hexadecimal values; and a high range of dividend values in the third region named dividend range FF, suggesting a range from six-bit values or lower two-digit hexadecimal values to full eight-bit values or higher two-digit hexadecimal values. The execution of the proposed divider is performed based on the USP-Awadhoot division algorithm by considering different divisor values starting from half-filled four-bit values or lower values of single hexadecimal digits to full eight-bit values or two-digit hexadecimal digit values. A relative presentation between the various combinations of operands with their required clock cycles is provided to compute the results.

The results generated from this clock performance analysis state that the proposed divider circuit based on the USP-Awadhoot algorithm requires a variable number of clock cycles to perform division operations on the particular operands provided at specific times. This shows that the proposed divider circuit requires the fewest clock cycles (0 clock cycles) when the operands exhibit invalid conditions. An invalid condition suggests that the divisor value is zero, and division by zero yields an indefinite condition that causes the generation of the invalid condition. An exception to the invalid condition exists with a dividend value of zero; when the input operand value indicates that the dividend and divisor values are both zero, then the proposed divider circuit takes slightly more clock cycles (17 clock cycles) to finish the execution process. The important reason behind this is that the circuit first detects of dividend value to indicate it as a nonzero value. If it detects a dividend value of zero, it sets the temporary output to zero and checks the divisor for a nonzero value. If it detects a nonzero value, then the final answer is zero. In a case with zero, the result must be changed to an invalid result generating an error signal, which requires extra clock cycles to execute the error signal generation process. The most clock cycles required for computing operands are two hundred and seventy-five clock cycles for the combination of the lowest divisor and the highest dividend value, indicating that more iterations must be completed before reaching the final result. When the divisor value is closer to the dividend value, the number of clock cycles required to execute the division step is lower.

Figure 8 illustrates the behaviour of the proposed divider, based on the difference between the dividend and divisor values. It is clear that the performance of the proposed divider is data-dependent, the number of average clock cycles required over more than half the range of the difference between the dividend and divisor falls into an almost fixed range of clock cycles. The required values of the average maximum numbers of clock cycles stay in the range of twenty-eight to sixty-three clock cycles. When the distance between the divisor value and dividend

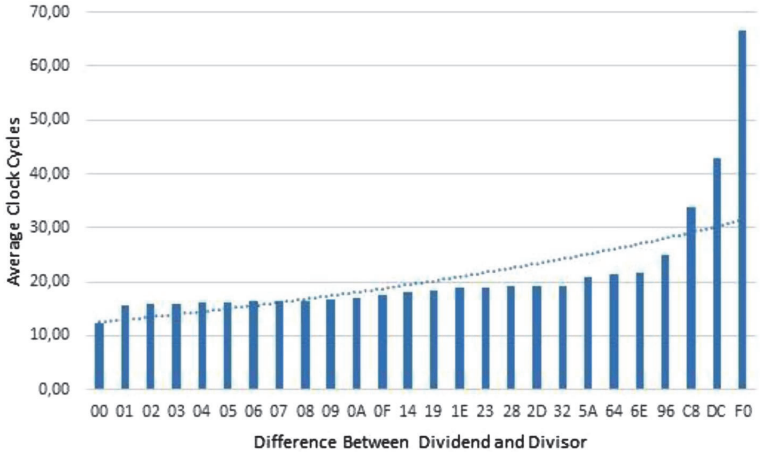


Figure 8. Clock performance analysis of the proposed USP-Awadhoot algorithm-based divider.

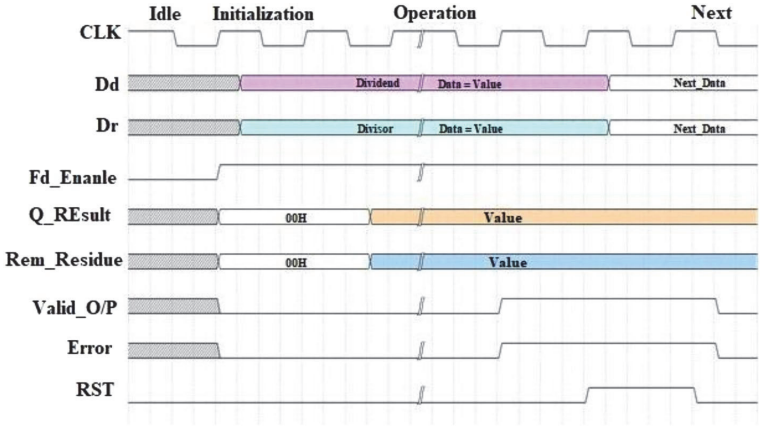


Figure 9. Waveform analysis of the proposed USP-Awadhoot algorithm-based divider.

value is less, the minimum required average number of clock cycles stays in the range of thirteen to twenty-four clock cycles. The lowest number of clock cycles (7 clock cycles) is required when the divisor value is unity.

Because the dividend value was previously confirmed to be a nonzero value and no iteration is performed, the final result value is calculated directly after confirming that the divisor value is unity. Midrange operand combinations possess clock requirements in the range of fifteen to thirty-five clock cycles. The divider circuit clock performance demonstrates the variable latency induced by the variable number of clock cycles required for different operand combinations in the proposed divider implementation and execution.

Waveform analysis. This section discusses the functional waveform analysis of the proposed divider based on the USP-Awadhoot division algorithm. We present the working conditions of the various signals used or generated by implementations in different situations, such as idle/initial and off/on states. We consider various combinations of dividends and divisors, giving better insight into different signals and data at the time of a particular conversion process. A waveform analysis is used to study the nine signals data to provide a clear idea regarding the proposed divider's working conditions based on the USP-Awadhoot division algorithm. The nine signals, such as the reference CLK, dividend (D_d), divisor (D_r), enable (F_d_enable), quotient (Q_Result), remainder (Rem_Residue), Valid_O/P, Error and RST required for waveform analysis, are distributed into five different groups depending on the nature of each signal: the reference group, I/P operand group, control group, O/P results group and indicator group.

Different dividend and divisor combinations require different clock cycles to operate; the reference group CLK signal provides the timing reference signal for computation execution. The reference clock signal's period

value is dependent on the working frequency; thus, the higher the frequency is, the lower the clock period value. The I/P operand group consists of the dividend (D_d) and the divisor (D_r) signals, which indicate the dividend and divisor values, respectively. The control group consists of F_d_*enable* and RST signals to provide start and end control for the computation process. The indicator group consists of Valid_O/P and Error signals, indicating computation completion and alerting the system of any invalid working condition or incorrect execution. The last and essential O/P results group consists of the Q_Result and Rem_Residue signals that provide the values of the quotient and remainder, respectively, as results of the division operation performed by the proposed divider based on the USP-Awadhoot algorithm. Figure 9 indicates a reference working waveform diagram concerning the initial working condition. The initial working waveform is mainly sectorized into an idle state, an initialization state, an operation state, and the next state. The idle state shows the proposed divider circuit's nonworking or stationary condition and beginning state when only a power supply is provided to the circuit after a shutdown. In the idle state, the CLK signal continues generating the reference signal, and the values of the I/P operand group's dividend (D_d) and divisor (D_r) signals are in the high-impedance tri-state condition. The control group $F_{d\,enable}$ and RST signals both possess low logic values, suggesting no operation. Similarly, the values of the indicator group and O/P result group's Q_Result and Rem_Residue signals are in a high-impedance tri-state condition, suggesting a stationary work condition. The initialization state indicates the next stage after applying the F_d_*enable* control signal to the proposed divider. In the initialization state, the proposed circuit resets the signal value of the O/P results group to the initial value of 00H, suggesting no result at the start. Later, it fetches the dividend (D_d) and divisor (D_r) data values from the input data lines that are stored in the input operand registers for further computation, as described in the previous sections. The indicator group's Valid_O/P and Error signals are set to logic low values, indicating that no computation operations have been performed yet.

The proposed divider circuit computes the Q_Result during the operation state, and the Rem_Residue signal provides the absolute value. After completing the computation operation, the Valid_O/P and Error are updated, and whether the computation and O/P result values are correct or incorrect is validated. At any instance, if a high-logic signal activates an RST signal, then the proposed divider circuit suspends its current computation operation state and resets it to the initialization state. The RST signal value is set to low logic, indicating an inactive reset signal that allows the continuing computation process to execute the division operation and compute the final O/P result values. Thus, depending on the control group signal values, the proposed divider circuit is ready to execute the next state.

Comparative statistics. Due to strict conversion rules, the division is the most complex essential arithmetic operation and is difficult to implement. As discussed in the division circuit block taxonomy section, many ways to identify different divider circuits are available. Given the continued growth of industry and technological improvements, there is a demand for achieving an efficient trade-off between the area, latency time, and criticality of the conversion logic. Table 1 illustrates different dividers based on their mathematical formulations and theoretical backgrounds. Digit recurrence-based dividers are the most commercially implemented divider circuits that provide many ideas, processes, and hardware architectures. However, much room remains to improve digit recurrence algorithm-based divider circuits' areas, frequencies, and power levels. We develop a new state-of-the-art digit recurrence algorithm-based divider named the USP-Awadhoot algorithm divider.

During various simulations performed on the Vivado 2016 simulation tool, the results of different input operand combinations are collected, forming a truth table for all possible combinations of input operands. The collected simulation data are summarized into time values, current operand (dividend and divisor) values, the present values of the control signals, and the outputs (quotient, remainder, error state, and valid output) obtained by the computation of the algorithm. The data collected from the output of the Xilinxs simulation and hardware implementation on Xilinxs FPGAs are compared with their known solutions from the reference truth table prepared from theoretical calculations. Once the data obtained from the output of the hardware implementation are successfully compared, the resources and execution speeds obtained with various divider implementations are determined. However, the hardware resource utilization data presented in Fig. 7 of the proposed USP-Awadhoot algorithm-based divider is considered further for comparative analysis.

Bailey[1] presented an article about statistical implementation data for restoring and nonrestoring algorithms in 2006. He presented a comparative analysis of the FPGA and Handel-C software implementation of restoring and nonrestoring division algorithms. These algorithms were implemented on RC-100 and RC-300 development boards produced by Celoxica using Xilinx's Spartan-II and a Virtex-II FPGA. A statistical comparison between the algorithms implemented as macro expressions with the Handel-C built-in integer divider is presented. For comparison, only restoring and nonrestoring algorithms based on the basic equations expressed in an earlier section are used without implementing the radix SRT algorithm.

The comparison presented in Table 2 concludes that Handel-C built-in divider is the slowest, as it can work on frequencies near 10 MHz. The chip area required in the FPGA is approximately more than double the area required by the proposed USP-Awadhoot algorithm-based divider. In Handel-C implementations, the use of subtraction for performing comparisons, its reuse as an input to a multiplexer, and the utilization of separate LUTs for addition and multiplexing requires extra hardware, limiting the speed improvement. The basic ideas of restoring and nonrestoring division algorithms can be implemented in a sufficiently small chip area, but the maximum working frequency is low. The number of LUTs may vary based on the hardware description languages (HDLs) used to implement the above algorithms.

In[5], Matthews et al. discussed integer divider designs for the ascendancy of FPGA-based soft-processor over the adaptation of variable-latency execution units in their instruction pipeline. The implementation efforts were focused on the Quick-Div divider, which exhibits data dependency and variable latency in integer division. This divider was integrated into the FPGA-based Taiga RISC-V pipelined soft processor.

Sr. no.	Algorithm	Equations	Important points
1	Restoring divider	For Jth 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 LUT
			Iterative subtraction is performed
			The nonredundant number system is used to write a quotient
			If the partial remainder value is not 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 the subtractor, shift register, and multiplier provide the approximate area requirement for algorithmic implementation
			Checks for possible MSB losses and overflow are needed
			Requires a full-width comparison at every iteration to obtain one bit of a quotient
			The quotient needs to be rearranged to obtain the actual quotient
2	Nonrestoring divider	For Jth 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 the restoring algorithm, it does not require restoring the partial remainder if the subtraction result is negative
			No requirement for a LUT
			The operation in each iteration depends on the result of the previous iteration
			Only one addition or subtraction operation can be performed in each iteration, so separate hardware is needed
			The partial remainder is kept between − Dr and + Dr, and the quotient digit is − 1 or 1
			It requires a sign bit to decide whether to perform addition or subtraction; the adder, subtractor, and shift register give the approximate area requirement for algorithmic implementation
			Requires an extra bit to be added with the partial remainder to have a track on a sign
			Requires a separate adder and subtractor in each iteration
			The area utilization of the implementation is approximately equal to the area required to implement the adder, subtractor, and shift register
3	SRT divider	For Jth iteration $q_j = \overline{1}$ if $2R_{j-1} < -D_r$ $q_j = 0$ if $-D_r \leq 2R_{j-1} \leq D_r$ $q_j = 1$ if $2R_{j-1} \geq D_r$ Has one of the values *-m, −m + 1… −1, 0, + 1. . . m−1, m*, where m is an integer comprising k digits of radix-n as $\frac{1}{2}(n-1) \leq m \leq n-1$ $n = 2^b$ and $k = x/b$ $Q = \sum_{j=1}^{k} q_j n^{-j}$ Quotient q is generated as a dividend through division by a divisor with the x most significant bits to remove b bits from the quotient in each iteration. Thus, radix-n performs k iterations to obtain the desired quotient	It is a nonrestoring algorithm based on radix-n
			Named after Dura W. Sweeney, James E. Robertson, and Keith D. Tocher
			For x bits, integer division requires k = x/b iterations, where b = the 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, and radix–n is typically selected as a power of base 2
			Each quotient digit has a value from {−m, −m + 1, ….., −1, 0, 1, ……., m−1, m}
			The algorithm implements 2's complement value of D_r instead of D_r. This enables shifting over zeros to eliminate extra adders and subtractors
			It needs an extra subtractor to determine the next partial remainder
			Error results are obtained due to few MSBs being used to predict the quotient bits (as in the low radix case); the error decreases with the increase in the radix
			A quotient selection table plus a carry-saving adder (CSA) gives the approximate area requirement for algorithmic implementation. It requires a quotient selection LUT. This shows the iteration time of accessing the quotient selection table plus multiple additions and subtractions
			Selecting higher quotient bits causes complexity in the quotient selection logic, and higher radix implementations are complex due to the impractical multiples of the divisor
			It needs to convert the last remainder to a conventional representation to find the sign bit, and the quotient correction stage selection depends on the sign bit

Continued

Sr. no.	Algorithm	Equations	Important points
4	Very high radix	*******	It removes more than ten quotient bits in one iteration; it requires a very large LUT with a large capacity for the quotient selection logic. A LUT is required for obtaining initial approximations of the reciprocal and the quotient digit selection logic
			It uses multiplication to form divisor multiples
			It differs from the regular radix-n divider in terms of the number and type of operations used in each iteration and the quotient digit selection logic
			A high radix makes quotient selection logic more complex and impractical to implement
5	Taylor series	$q = D_d/D_r \; and \; X_0 = 1/D_r$ $q = D_d X_0\{1 + (1 - D_r X_0) + (1 - D_r X_0)^2 + (1 - D_r X_0)^3\}$ $D_d =$ Dividend and $D_r =$ Divisor $1/D_r = Antidivisor$	It is a multiplicative iteration-based algorithm that thus requires a large area
			The precision depends upon the closeness with the antidivisor (reciprocal) estimation
			It provides a parallel powering section that computes high-order terms faster, with a minimal extension of the hardware overhead
			A quotient digit selection logic LUT and a three-full-word-length multiplier give the approximate area requirement for algorithmic implementation
6	Newton–Raphson	$Q = D_d/D_r = p \times (q)^{-1}$ $f(X) = 1/X - q^{-1} = 0$ $X_{i+1} = X_i - \frac{f(X_i)}{f'(X_i)}$ $X_{i+1} = X_i - \frac{(1/X_i - q^{-1})}{1/X_i^2} = X_i \times (2 - q^{-1} \times X_i)$ $\in_{i+1} = \in_i^2 (q^{-1})$ $p =$ Dividend and $(q)^{-1} = Antidivisor$	The accuracy can be improved by selecting a proper root at the beginning
			The latency and error during convergence are directly dependent on the root selected at the beginning of the convergence process, the iteration time is approximately equal to the time required for two serial multiplications
			A multiplier, a quotient selection LUT, and control logic give the approximate area requirement for algorithmic implementation
			The final quotient is derived by multiplying the approximated reciprocal and dividend
			Shows the error induced due to the inaccuracy of the quotient digit prediction or estimation
			It requires multiplication and addition or subtraction at each iteration, and using 1's complement induces more error
7	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 the dividend and divisor by the antidivisor 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 delays, but this adds a new approximation error in each iteration
			A quotient digit selection logic LUT, a one-full-word-length multiplier, and a one-full-word-length adder/subtractor logic give the approximate area requirement for algorithmic implementation
8	Variable-latency	******	Its variable execution time thus results in different conversion times for different sets of dividends and divisors
			Self-timing, result caching, and quotient digit speculation are some techniques used to provide variable latency
8	Variable latency	******	The DEC Alpha 21,164 is one of the best variable-latency class algorithm implementation examples, based on the concepts of the simple normalizing and nonrestoring division algorithm
9	Svoboda algorithm and Svoboda-Tung algorithm	$\left\{\frac{mn}{(m+1)(n-1)} < D_r < \frac{m(n-2)}{(n-1)(m-1)}\right\}$ $\{-m/n - 1 < R_j < m/n - 1\}$ $Range = \{0, \pm 1, \ldots \ldots, \pm m\}$ $Boundry\ limit = \{n/2 + 1 \le m \le n - 1\}$ $m = Range\ of\ SBD\ and\ n = Radix$	The quotient digit is predicted based on the partial remainder without considering the divisor; one or two MSBs of the partial remainder are used for generating quotient digit selection logic
			It can select a quotient digit out of the radix range if an overflow occurs due to compensation
			It requires prescaled operands and can work on conventional and signed digit ranges
			It is also a radix-n based algorithm with signed binary digit numbers, making it similar to the SRT algorithm
			It is applicable more than radix 4, and Prescaled operands are needed; it needs extra multipliers, resulting in more hardware overhead
Continued			

Sr. no.	Algorithm	Equations	Important points
10	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., a dividend bit count of 4n and divisor bit counts of n
			We can represent dividends in terms of fixed partitions based on their associated weights as per the dividers' radix values
			The area is directly dependent on the number of dividend partitions related to the dividers' radix values
11	Jebelean exact division	$D_d = d * Q$ $D_d = D_{dupk} n^k + D_k$ $b_k = \left(-n^{-k} D_{dk}\right)_{modd}$ $b_k = \left(-n_{modd}^{-k} D_{dk}\right)_{modd}$ $modPower\ calls\ and\ ParallelPrefixSum\ call = O(logn)$	It is applied when complete division is performed on long integer operands in digital computation, even after knowing that the remainder is zero
			It starts from the least significant digits of the operands
			Remarkable performance is observed when the radix is a prime or power of 2
			It takes constant execution time to access a fixed-word-length LUT
			It takes O(log n) execution time, and for short division, $O(n/\rho + \log\rho)$, where n is the word length of the dividend and ρ is the number of processors
			It needs synchronization to execute calculations in parallel

Table 1. Summary of a comparative study on different division algorithm-based dividers.

Parameter/result	Slice logic LUTs	Clock frequency (MHz)	
		From	To
Handel–C	747	721	10,965
Restoring	115	13,716	20,345
Nonrestoring	144	24,175	40,073
Nonrestoring with pipeline	66	37,806	63,558
Proposed USP-Awadhoot divider version V8.9	266	50	285

Table 2. Comparative analysis of the resource utilization of the proposed USP-Awadhoot algorithm-based divider with those of the Handle-C and digit recurrence algorithm-based dividers.

Name/parameter	Slice logic LUTs	Slice register flip-flop	Clock 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
Proposed USP-Awadhoot divider—8 bits version V8.9	266	146	285
Proposed USP-Awadhoot divider—16 bits version V16.1	622	241	125

Table 3. Comparative analysis of the resource utilization of the proposed USP-Awadhoot algorithm-based divider with variable-latency Quick-Div dividers and fixed-latency radix-n dividers.

Taiga is a RISC-V open-source soft processor. Experimental implementations are performed over the Xilinx Virtex UltraScale + VCU118 board (XCVU9P-L2FLGA2104E) using Vivado 2018.3 synthesis. With ascendancy over the variable-latency execution unit's operation 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 implementations of the data-dependent variable-latency Quick-Div dividers and fixed- latency radix-n (n = 2, 4, 8, 16) dividers with the RISC-V Taiga soft processor. Quick-Div dividers are unsigned processes, so sign conversion is performed before and after conversion; completion is required depending on the instruction operands and types. Therefore,

Name/parameter	Slice logic LUTs	Slice register flip-flops	Look-up tables	Frequency (MHz)
IP core-pipelined divider 8-bit	2247	4020	1400	204.3
IP core-pipelined divider 16-bit	2742	4904	1680	201.6
IP core-pipelined divider 32-bit	3843	6864	2240	193.1
Proposed USP-Awadhoot divider—8 bits version V8.9	266	146	0	285
Proposed USP-Awadhoot divider—16 bits version V16.1	622	241	0	125

Table 4. Comparative analysis of the resource utilization of the proposed USP-Awadhoot algorithm-based divider with that of the Xilinx IP core-pipelined divider.

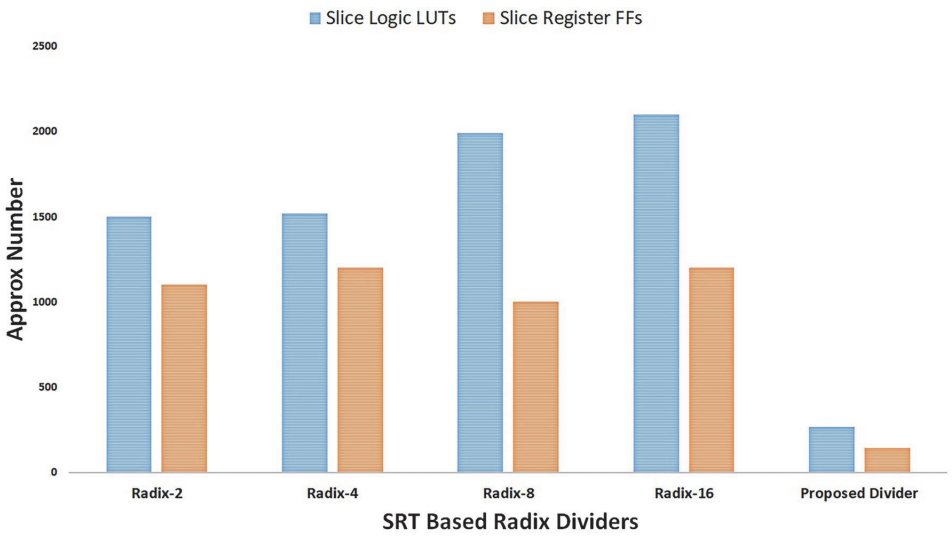


Figure 10. Comparative analysis regarding the hardware resource utilization of the proposed USP-Awadhoot algorithm-based divider and radix-n-based dividers.

Quick-Div requires an additional three cycles for sign conversion. Table 3 illustrates the comprehensive results of the variable-latency Quick-Div dividers and the fixed-latency radix-n (n = 2, 4, 8, 16) dividers with the RISC-V Taiga soft processor compared with those of 8/16-bit USP-Awadhoot dividers. This indicates that the variable-latency Quick-Div dividers and fixed-latency radix-n (n = 2, 4, 8, 16) dividers require 5 to 7 times more chip area than the proposed USP-Awadhoot divider; this is represented by the numbers of slice logic LUTs and slice register flip-flops used for the implementation. In contrast, the maximum clock frequency is almost double the maximum clock frequency of the proposed USP-Awadhoot divider.

In[9], Sorokin discussed the implementations of fixed-point dividers based on different algorithms on Xilinx's common FPGA platform. Different divider modules have been compared with Xilinx's 32-bit IP core-pipelined divider. This indicates that a nonrestoring algorithm-based fixed-point divider module is much faster than the 32-bit Xilinx IP core-pipelined divider. This paper points out that the results are more approximations than exact values and demonstrate more practical division operations than digital operations. These approximated values can cause trouble in more critical applications, such as biomedical applications, sensor signal processing, coordinate computation for an item, etc.[9]. As we have discussed earlier, even for integer division, we must use a 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[78]. 32-bit input operands produce 32-bit remainders in many cases, making them impossible to implement in applications where high calculation precision is needed. Another implementation-focused problem in this article concerns the chip area requirements of this solution. The fixed-point algorithm follows the basic principles of the simple paper-and-pencil division algorithm. A fixed-bit-length quotient is generated in every iteration of a fixed-point divider, similar to digit recurrence dividers. Much attention is given to improving the addition and multiplication operations, as speeding up addition operations reduces the computational time required in the actual division process. Replacing the divisor with its inverse value can allow the use of multiplication by an antidivider to obtain division results.

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, such as the availability of drop-in modules for

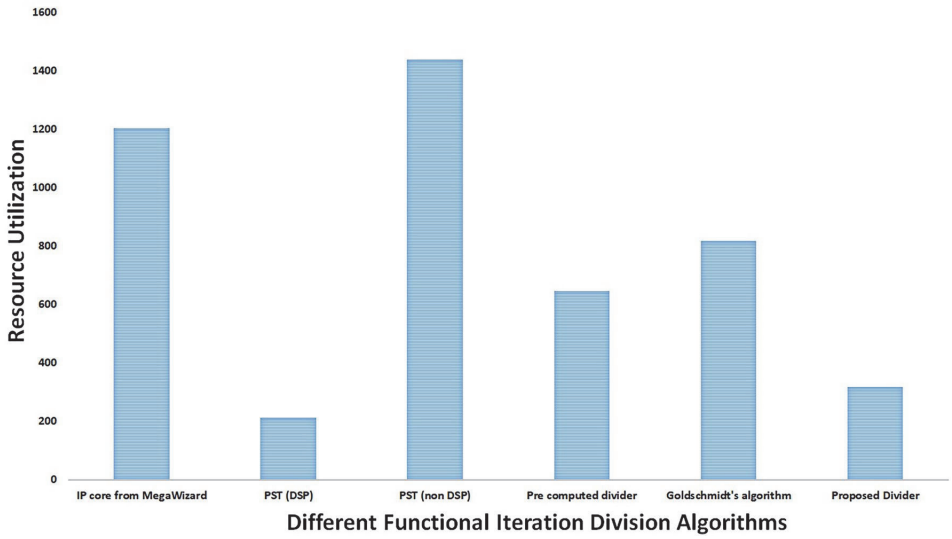


Figure 11. Comparative analysis regarding the hardware resource utilization of the proposed USP-Awadhoot algorithm-based divider and different functional dividers.

Virtex, Virtex-II, Virtex-II Pro, Virtex-4, Spartan-3, etc. The dividend can be up to 32 bits and has a fully pipelined structure. Table 4 compares 8/16/32-bit Xilinx IP core-pipelined dividers and the proposed USP-Awadhoot divider. This indicates that Xilinx's IP core-pipelined dividers are bulkier, with three to five times more chip area, and the maximum clock frequency is almost the same as that of the proposed USP-Awadhoot dividers.

Many nonrestoring algorithms have been designed and implemented, but the SRT algorithm is the most implemented approach. The basic SRT algorithm was implemented in[5,9,12,18,20,25,42,51,62,68,76–83] for different applications utilizing different aspects of the algorithm. Figure 10 illustrates the comparative analysis regarding the hardware resource utilization of the USP-Awadhoot division algorithm-based divider and other SRT-based radix-n dividers. On average, the proposed USP-Awadhoot algorithm-based divider requires 266 slice logic LUTs, 146 slice register flip-flops with a power dissipation estimation of 3.366 watts. In contrast, the radix-2 to radix-16 divider implementations require 1500 to 2100 slice logic LUTs and 1100 to 1200 slice register flip-flops[5]. This indicates that the concept of different prescaling factors for the input operands used in the proposed USP-Awadhoot divider helps reduce its chip area requirements.

From the commercial and noncommercial implementation points of view, two classes of dividers are the main focus. One is the famous digit recurrence class, and the other is the functional iteration class of dividers. In[6], Tatas et al. discussed different concepts involved in partitioning the main dividend into segments to represent an actual division of a numerator by a denominator as a series of smaller divisions with necessary requirement for numerator to meet (Numerator N= N1 + N2 +...). This concept of a series of divisions showcases a smaller-dividend division algorithm, where we must perform shifting, partial division, and accumulation operations. All intermediate operations are performed by considering the weights of the dividend bits.

$$\frac{N}{D} = \frac{N1}{D} + \frac{N2}{D} + \frac{N3}{D} + \frac{N4}{D} + \cdots \tag{32}$$

This algorithm can be implemented in both series and parallel architecture[8], but a higher-radix system is critical and difficult to implement. The partitioned numerator's partial division process can be performed serially or in parallel due to the trend between cost and time. This algorithm is implemented with a length of N, D 32-bit dividends and a parallel-array divider, a sequential divider with two partitions, or a parallel divider with two partitions in the partial division stage. These implementations require 4316, 2136, and 3050 slices on a Xilinx Virtex-E 1000. From the above data, it is clear that the sequential implementation of the proposed algorithm is more area-efficient and moderate in terms of the time delay. If any corrective stage is required in a sequential divider, this will degrade the efficiency of serial dividers. In contrast, parallel implementation produces a slight reduction in the delay but insufficient decreases in the area and latency. The array implementation of this algorithm is inefficient as it increases the chip area by four times based on doubling the word length. Whereas the proposed USP-Awadhoot algorithm-based divider does not partition the given Numerator (Dividend) into smaller dividends like mentioned above (Numerator N = N1 + N2 +...). The USP-Awadhoot algorithm-based divider converts the Numerator (Dividend) into group dividends, which are not required to add up to the main Dividend value. The proposed USP-Awadhoot algorithm-based divider requires 266 slice logic LUTs, 146 slice register flip-flops with a power dissipation estimation of 3.366 watts indicating better implementation area performance.

Parameter/result	Case 1	Case 2	Case 3	Case 4	Proposed divider version V8.9
Dividend width	8	8	8	8	8
Divisor width	8	8	8	8	8
Remainder and quotient width	8	8	8	8	8
LUT6-FF pairs	223	218	217	215	000
Slice logic LUTs	203	205	203	197	266
Slice register flip-flops	288	288	288	288	146
IC name	Virtex7	Kintex7	Virtex6	Spartan6	Xilinx Zynq XC7Z010

Table 5. Comparative analysis regarding the resource utilization of the proposed USP-Awadhoot algorithm-based divider and Xilinx's LogiCORE IP divider generator V4.0.

In[27], Kasim et al. discussed a divider block with precomputed values stored in ROM in terms of a LUT. This divider operation is similar to the dividers based on functional iteration algorithms such as Goldschmidt's algorithm and Newton's method[27]. The result of this divider is also an approximate value, unlike those of iterative subtraction class-based dividers. In[38], Liu et al. discussed an algorithm that utilizes prescaling, series expansion, and Taylor series expansion together; hence, it is sometimes called a phase stretch transform (PST) algorithm. At the start, both operands are prescaled up to the suitable starting level. Operand prescaling is performed based on a scaling factor E0, stored in a LUT. In the second stage of the PST algorithm, series expansion is applied to the scaled operands to obtain an accurate antidivisor approximation. To calculate the partial quotient and the next remainder in the iteration stage, it utilizes 0-order Taylor series expansion. The iterative process must continue until a quotient is obtained with the required precision range of error. Three Taylor expansion iterations and a LUTs are needed to finish one operation.

Figure 11 illustrates a comparative analysis regarding the hardware resource utilization of the proposed USP-Awadhoot algorithm-based divider implementation and that of different functional iteration divider implementations, as discussed above. As per the performance comparison between the proposed USP-Awadhoot divider and the Mega Wizard IP core, DSP, and non-DSP structures of the divider algorithm, the resource utilization of the proposed USP-Awadhoot algorithm-based divider includes 266 slice logic LUTs, 146 slice register flip-flops with a power dissipation estimation of 3.366 watts; whereas the PST algorithm-based divider requires 213 slice logic LUTs, 768 bytes of memory, and 28 DSP[84], the PST algorithm-based divider without DSP needs 1437 slice logic LUTs and 768 bytes of memory. The precomputed divider and Goldschmidt's algorithm-based divider require 647 and 816 slice logic LUTs, respectively. The divider algorithm's Mega Wizard IP core, DSP, and non-DSP structures significantly delay the results, as their maximum clock frequencies are limited to 50 to 73 MHz. This framework does not save sufficient area relative to the proposed USP-Awadhoot divider. Additionally, the precomputed values introduce rounding errors in the calculation process. The proposed USP-Awadhoot divider displays better implementation area requirements and maximum clock frequency performance.

Table 5 illustrates a comparative analysis regarding the resource utilization of the proposed USP-Awadhoot division algorithm-based divider and the Xilinx LogiCORE IP Divider Generator V4.0[62]. The proposed USP-Awadhoot algorithm-based divider requires 266 slice logic LUTs, 146 slice register flip-flops, and a total of 37 bounded I/o's for providing input operands and reading the quotient and remainder outputs. The operating speed of the proposed divider is given in terms of its operating frequency, which is equal to 285 MHz, with a power dissipation estimation of 3.366 watts. The implementation statistics are derived for the proposed divider with a dividend width of eight bits, a divisor width of eight bits, a quotient width of eight bits, a remainder width of eight bits, and Error and Valid_O/P signals. The Valid_O/P signal indicates the computation's completion, whereas the Error signal indicates an invalid condition caused by a divisor value of zero, i.e., the divide-by-zero condition. The resource utilization analysis of the proposed divider implementation is conducted with Xilinx's LogiCORE IP Divider Generator V4.0. Xilinx is the leading candidate in the IC industry and has a wide range of Intellectual property (IPs). We consider Virtex 6 and 7, Kintex 7, and the Spartan 6 FPGA from Xilinx during the comparison. The number of slice register flip-flops used is constant at 288 for each FPGA IC, whereas the number of slice logic LUTs used ranges from 197 to 205 and the number of input LUT-FF pairs used slightly varies from 223 to 215 for Virtex 7, Kintex 7, Virtex 6 and Spartan 6. The document does not mention the power consumption of the LogiCORE IP Divider Generator V4.0, whereas the proposed divider based on the USP-Awadhoot division algorithm simulation estimates 3.366 Watts.

The proposed USP-Awadhoot algorithm-based divider implementation utilizes 64% less FPGA hardware resources than the Handel-C built-in divider but, it requires 57% and 75% more FPGA hardware resources than simple restoring and nonrestoring dividers. The proposed divider's working frequency based on the USP-Awadhoot algorithm implementation versions is approximatly 50% greater than other performances presented in[1], indicating the FPGA resource utilization improvement achieved by the proposed USP-Awadhoot algorithm-based divider implementation. Based on the statistics presented in[9], the proposed USP-Awadhoot algorithm-based divider implementation shows improvements in its FPGA resource utilization in terms of 86% to 88% improvements in the number of required slice logic LUTs (depending on the use of 8-bit or 16-bit operands) and 95% to 96% improvements in the number of slice register flip-flops required (depending on the use of 8-bit or 16-bit operands). Based on the statistics presented in[5], the proposed USP-Awadhoot algorithm-based 8-bit divider implementation shows improvement in slice logic LUTs and slice register flip-flops requirement as the

proposed divider implementation do not require any six input LUT-FF pairs. As compared to the variable-latency Quick-Div dividers and fixed-latency radix-n dividers but the results exhibit a comparatively lower working frequency of 285 MHz for the proposed divider. The power required for variable-latency Quick-Div dividers and fixed latency radix-n dividers is not mentioned, but the proposed divider simulation estimates 3.366 Watts.

As per the comparative statistics presented in[27], the proposed USP-Awadhoot algorithm-based divider implementation exhibits an FPGA resource utilization improvement in terms of a 58% improvement in the number of required slice logic LUTs compared to that of the precomputed divider, a 67% improvement in the number of required slice logic LUTs compared to that of Goldschmidt's algorithm-based divider, a 76% improvement in the number of required slice logic LUTs compared to that of the divider developed from Quartus Mega functions. The proposed divider does not induce any errors in the computed results and simulations estimated required power of 3.366 watts. As per the statistics presented in[38] It achieves a 82% improvement over the IP core from MegaWizard's operating frequency as it is upto 50.16 MHz and a 75% improvement over the PST-DSP and non-DSP dividers as it is upto 73 MHz. The proposed divider requires no extended memory or DSP compared to the IP core from the MegaWizard and the PST divider.

Conclusion

The evaluation of addition and multiplication implementations typically falls into the latency range from a couple of clock cycles to less than ten clock cycles, while the performance evaluation of division operation implementations typically falls into the latency range from tens to hundreds of clock cycles and requires a high implementation area. The primary focus of the problem statement is to design and implement a reduced-area divider circuit block, providing straightforward dialectics between divisors, dividends, and quotients to avoid round-off errors. A design is developed by simulating the proposed technique and cross-verified by performing regular sequential and pseudorandom sequential analyses of the implementation against standard result tables generated by simulations and the theoretical study of the proposed idea. The main contributions highlighted in the article are as follows.

- Significant efforts were taken toward developing the state-of-the-art novel USP-Awadhoot algorithm-based divider circuit block implementation.
- The proposed USP-Awadhoot algorithm-based divider circuit block implementation substantially reduces the required implementation resources, resulting in better area efficiency.
- Successful implementation of a dynamic separate scaling operation/factor for input operands to reduce the conversion complexity.
- A novel divisor-dividend relationship is demonstrated with the proposed USP-Awadhoot algorithm-based divider circuit block implementation to derive dividend groups, modified divisors (MD_r), and FD terms. This proves that the hypothesis of utilizing different scaling operation/factors for dividends and divisors can improve the area requirements by reducing resource utilization.
- A comparatively straightforward group quotient (GQ_n) value selection logic is developed based on the unique relations derived between the dividend groups, modified divisors (MD_r), and FD terms of the proposed technique or algorithm of the divider circuit block implementation; this is termed the Awadhoot matrix.
- A comparatively straightforward process for selecting the final quotient based on the group quotient (GQ_n), partial quotient (PQ_n), and additional quotient (AQ) values is developed.

The second-most significant contribution is the design and verification of complex division via the Baudhayan-Pythagoras triplet method using the novel state-of-the-art USP-Awadhoot divider circuit block implementation.

Future work road map

- As the current implementation verifies the successful implementation of the proposed divider on different FPGAs, the next target is to design a dedicated integrated-circuit IP. The first step is to design a physical layout, starting from the floor plan, which determines which circuit component is placed in which area and extracts the parasitic values to prepare the final layout for fabrication.
- Another future work target is to improve the working frequency and conversion time. To do so, we must fuse some intermediate functional blocks such as separate addition, and multiplication can be performed in the fused mode such as fused-multiply-add (FMA). We need to test the implementation and verify the resource utilization of the proposed divider to validate these changes.
- The current implementation validates its successful implementation using combinational circuits. Thus, reducing the area and hardware resource utilization is also a future target. Some processes involved in the proposed divider can be represented as different hardware architectures, such as pipelined architectures, parallel architectures, array structures, and cascade structures. Thus, it is necessary to validate the usage of different architectures and compare their resource utilization levels to prove the usability of the proposed divider in different working environments with different requirements. Detailed implementation results will be utilized to choose the most suitable architecture implementation of the proposed divider in various applications based on their time, area, and power requirements.
- We must verify the performance of the proposed divider in various applications, such as image processing, particle detection, and signal processing. Complex number arithmetic is critical and requires a careful design and more hardware resources. It is also very helpful in various essential engineering applications, such as

acoustic pulse reflectometry, astronomy, nonlinear radio frequency measurements, control theory applications such as finding root loci, Nyquist plots, and microwave system frequency responses.

Methods

Process of FPGA-based circuit integration. The circuit constructed based on the proposed USP-Awadhoot division algorithm is preliminarily implemented on an FPGA, and the computational results are studied and analyzed with the help of FPGA design simulation suites. We utilize the Xilinx Vivado design suite and the Altera Quartus II design suite for the proposed design. Before utilizing the design suites, we subdivide the complete computation into different FSM states. VHDL is utilized for the implementation, making it easier in design and performance simulations to analyze the resulting behavior. The tests and implementations of the circuits using FPGAs are parts of the design process for application-specific integrated circuit (ASIC) design. The manufacturing of the USP-Awadhoot division algorithm implemented in an ASIC is possible. We develop the proposed circuit implementation with the algorithm's established flow, which is tested in behavioral and implementation timing simulations to verify its performance and results. The post-simulation circuit is tested in the real world by implementing it in different FPGAs, especially those from Altera and Xilinx.

Data availability

All data generated or analyzed during this study are included in this published article [and its supplementary information files].

Received: 5 February 2022; Accepted: 17 January 2023
Published online: 21 February 2023

References

1. Bailey, D. G. Space-efficient division on FPGAs in Electronics New Zealand conference 206–211 (2006).
2. Qasaimeh, M. *et al.* Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels. In *2019 IEEE International Conference on Embedded Software and Systems (ICESS)* 1–8 (IEEE, 2019).
3. Kumari, J. & Yasin, M. Y. Design and soft implementation of N-bit SRT divider on FPGA through VHDL. *Int. J. Innov. Eng. Sci. Manag.* **3**, 13–19 (2015).
4. Narendra, K., Ahmed, S., Kumar, S. & Asha, G. FPGA implementation of fixed point integer divider using iterative array structure. *Int. J. Eng. Tech. Res.* **3**, 170–179 (2015).
5. Matthews, E., Lu, A., Fang, Z. & Shannon, L. Rethinking integer divider design for FPGA-based soft-processors. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* 289–297 (IEEE, 2019).
6. Tatas, K., Soudris, D. J., Siomos, D., Dasygenis, M. & Thanailakis, A. A novel division algorithm for parallel and sequential processing. In *9th International Conference on Electronics, Circuits and Systems* 553–556 (IEEE, 2002).
7. Tocher, K. D. Techniques of multiplication and division for automatic binary computers. *Q. J. Mech. Appl. Math.* **11**, 364–384 (1958).
8. Asai, H. A recursive radix conversion formula and its application to multiplication and division. *Comput. Math. Appl.* **2**, 255–265 (1976).
9. Sorokin, N. Implementation of high-speed fixed-point dividers on FPGA. *J. Comput. Sci. Technol.* **6**, 8–11 (2006).
10. Huang, K. & Chen, Y. Improving performance of floating point division on GPU and MIC. In *Algorithms and Architectures for Parallel Processing* (eds Wang, G. *et al.*) 691–703 (Springer International Publishing, 2015).
11. Fang, X. & Leeser, M. Vendor agnostic, high performance, double precision Floating Point division for FPGAs. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)* 1–5 (IEEE, 2013).
12. Liu, W. & Nannarelli, A. Power dissipation challenges in multicore floating-point units. In *ASAP 2010-21st IEEE International Conference on Application-Specific Systems, Architectures and Processors* 257–264 (IEEE, 2010).
13. Thall, A. *Extended-Precision Floating-Point Numbers for GPU Computation in ACM SIGGRAPH 2006 Research Posters 52-es* (Association for Computing Machinery, 2006).
14. Sinha, P. *Smart Sensors Use DSCs for Embedded Signal Processing* (Microchip Technology Inc., 2021).
15. Trummer, R. K. L. A high-performance data-dependent hardware integer divider. Master thesis (Institute of Computer Science and Systems Analysis, 2005).
16. Obermann, S. F. & Flynn, M. J. Division algorithms and implementations. *IEEE Trans. Comput.* **46**, 833–854 (1997).
17. Pineiro, A., Bruguera, J. D., Lamberti, F. & Montuschi, P. A radix-2 digit-by-digit architecture for cube root. *IEEE Trans. Comput.* **57**, 562–566 (2008).
18. Takagi, N., Kadowaki, S. & Takagi, K. A hardware algorithm for integer division. In *17th IEEE Symposium on Computer Arithmetic (ARITH'05)* 140–146 (IEEE, 2005).
19. Lee, B. R. & Burgess, N. Improved small multiplier based multiplication, squaring and division. In *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003*, 91–97 (IEEE, 2003).
20. Nannarelli, A. & Lang, T. Low-power divider. *IEEE Trans. Comput.* **48**, 2–14 (1999).
21. Nikmehr, H. Architectures for floating-point division. PhD thesis (School of Electrical and Electronic Engineering, The University of Adelaide, 2005).
22. Soderquist, P. & Leeser, M. Division and square root: choosing the right implementation. *IEEE Micro* **17**, 56–66 (1997).
23. Piso, D., Pineiro, J. A. & Bruguera, J. D. Analysis of the impact of different methods for division/square root computation in the performance of a superscalar microprocessor. In *Proceedings Euromicro Symposium on Digital System Design. Architectures, Methods and Tools* 218–225 (IEEE, 2002).
24. Detrey, J. & de Dinechin, F. A Tool for unbiased comparison between logarithmic and floating-point arithmetic. *J. VLSI Signal Process. Syst. Signal Image Video Technol.* **49**, 161–175 (2007).
25. Sutter, G., Bioul, G. & Deschamps, J.-P. Comparative study of SRT-dividers. In *FPGA in Field Programmable Logic and Application* (eds Becker, J. *et al.*) 209–220 (Springer, 2004).
26. Dixit, S. & Nadeem, M. FPGA accomplishment of a 16-bit divider. *Imp. J. Interdiscip. Res.* **3**, 140–143 (2017).
27. Kasim, M. F., Adiono, T., Fahreza, M. & Zakiy, M. F. FPGA implementation of fixed-point divider using pre-computed values. *Procedia Technol.* **11**, 206–211 (2013).
28. Oberman, S. F. & Flynn, M. J. *An Analysis of Division Algorithms and Implementations, Technical Report CSL-TR-95-675* (Stanford University, 1995).
29. Hongal, R. S. & Anita, D. Comparative study of different division algorithms for fixed and floating point arithmetic unit for embedded applications. *Int. J. Comput. Sci. Eng.* **4**, 48–54 (2016).

30. Schwarz, E. M. & Flynn, M. J. *Using a Floating-Point Multiplier's Internals for High-Radix Division and Square Root, Technical Report CSL-TR-93-554* (Stanford University, 1993).
31. Anderson, S. F., Earle, J. G., Goldschmidt, R. E. & Powers, D. M. The IBM system/360 model 91: Floating-point execution unit. *IBM J. Res. Dev.* **11**, 34–53 (1967).
32. Fowler, D. L. & Smith, J. E. An accurate, high speed implementation of division by reciprocal approximation. In *Proceedings of 9th Symposium on Computer Arithmetic* 60–67 (IEEE, 1989).
33. Liebig, B. & Koch, A. Low-latency double-precision floating-point division for FPGAs. In *2014 International Conference on Field-Programmable Technology (FPT)* 107–114 (IEEE, 2014).
34. Han, K.-N., Tenca, A. F. & Tran, D. High-speed floating-point divider with reduced area. In *Mathematics for Signal and Information Processing* 219–226 (SPIE, 2009).
35. Kong, I. & Swartzlander, E. E. A Goldschmidt division method with faster than quadratic convergence. *IEEE Trans. Very Large Scale Integr. Syst.* **19**, 696–700 (2011).
36. Goldschmidt, R. E. Applications of division by convergence. Masters Degree Thesis (Massachusetts Institute of Technology, 1964).
37. Kwon, T.-J., Sondeen, J. & Draper, J. Floating-point division and square root using a Taylor-series expansion algorithm. In *2007 50th Midwest Symposium on Circuits and Systems* 305–308 (IEEE, 2007).
38. Liu, J., Chang, M. & Cheng, C.-K. *An Iterative Division Algorithm for FPGAs* (Association for Computing Machinery, 2006).
39. Kumar, A. & Sasamal, T. N. Design of divider using taylor series in QCA. *Energy Procedia* **117**, 818–825 (2017).
40. Liddicoat, A. A. & Flynn, M. J. High-performance floating point divide. In *Proceedings Euromicro Symposium on Digital Systems Design* 354–361 (IEEE, 2001).
41. Bannon, P. & Keller, J. Internal architecture of Alpha 21164 microprocessor. In *Digest of Papers. COMPCON'95. Technologies for the Information Superhighway* 79–87 (IEEE, 1995).
42. Cortadella, J. & Lang, T. High-radix division and square-root with speculation. *IEEE Trans. Comput.* **43**, 919–931 (1994).
43. Lang, T. & Nannarelli, A. A radix-10 digit-recurrence division unit: algorithm and architecture. *IEEE Trans. Comput.* **56**, 727–739 (2007).
44. Sarma, D. D. & Matula, D. W. Faithful bipartite ROM reciprocal tables. In *Proceedings of the 12th Symposium on Computer Arithmetic* 17–28 (IEEE, 1995).
45. Tenca, A. F. & Ercegovac, M. D. On the design of high-radix on-line division for long precision. In *Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No.99CB36336)* 44–51 (IEEE, 1999).
46. Ugurdag, H. F., Dinechin, F. D., Gener, Y. S., Gören, S. & Didier, L. Hardware division by small integer constants. *IEEE Trans. Comput.* **66**, 2097–2110 (2017).
47. Mehta, B., Talukdar, J. & Gajjar, S. High speed SRT divider for intelligent embedded system. In *2017 International Conference on Soft Computing and Its Engineering Applications (icSoftComp)* 1–5 (IEEE, 2017).
48. Boullis, N. & Tisserand, A. On digit-recurrence division algorithms for self-timed circuits. In *Advanced Signal Processing Algorithms, Architectures, and Implementations XI* 115–125 (SPIE, 2001).
49. Parhami, B. *Computer Arithmetic: Algorithms and Hardware Designs* (Oxford University Press, 2010).
50. Ercegovac, M. D. & Lang, T. *Digital Arithmetic a Volume in the Morgan Kaufmann Series in Computer Architecture and Design* (Morgan Kaufmann, 2004).
51. Jun, K. Modified non-restoring division algorithm with improved delay profile. M.S. thesis (The University of Texas, 2011).
52. Robertson, J. E. A new class of digital division methods. *IRE Trans. Electron. Comput.* **EC-7**, 218–222 (1958).
53. Cocke, J. & Sweeney, D. W. *High-Speed Arithmetic in a Parallel Device, Technical Report* (IBM Corp., 1957).
54. Nadler, M. A high-speed electronic arithmetic unit for automatic computing machines. *Acta Tech.* **6**, 464–478 (1956).
55. Macsorley, O. L. High-speed arithmetic in binary computers. *Proc. IRE* **49**, 67–91 (1961).
56. Wilson, J. B. & Ledley, R. S. An algorithm for rapid binary division. *IRE Trans. Electron. Comput.* **EC-10**, 662–670 (1961).
57. Metze, G. A Class of binary divisions yielding minimally represented quotients. *IRE Trans. Electron. Comput.* **EC-11**, 761–764 (1962).
58. Montuschi, P. & Ciminiera, L. Simple radix 2 division and square root with skipping of some addition steps. In *[1991] Proceedings 10th IEEE Symposium on Computer Arithmetic* 202–209 (IEEE, 1991).
59. Mandelbaum, D. M. A systematic method for division with high average bit skipping. *IEEE Trans. Comput.* **39**, 127–130 (1990).
60. Atkins, D. E. *The Theory and Implementation of SRT Division, Technical Report UIUCDCS-R-67-230* (The University of Illinois, 1967).
61. Montuschi, P. & Ciminiera, L. Reducing iteration time when result digit is zero for radix 2 SRT division and square root with redundant remainders. *IEEE Trans. Comput.* **42**, 239–246 (1993).
62. Harris, D. L., Oberman, S. F. & Horowitz, M. A. SRT division architectures and implementations. In *Proceedings 13th IEEE Symposium on Computer Arithmetic* 18–25 (IEEE, 1997).
63. Sumiksha, K. P. & Shetty, S. Computation of SRT and CORDIC division algorithms. *IOSR J. Electron. Commun. Eng.* **12**, 53–56 (2017).
64. Clarke, E. M., German, S. M. & Zhao, X. *Verifying the SRT Division Algorithm Using Theorem Proving Techniques* (Springer, 1996).
65. Bryant, R. E. Bit-level analysis of an SRT divider circuit. In *33rd Design Automation Conference Proceedings, 1996* 661–665 (IEEE, 1996).
66. MicroBlaze processor reference guide, Xilinx Inc, preprint at http://xilinx.com/support/documentation/swmanuals/xilinx20164/ug984-vivado-microblaze-ref.pdf.
67. Jamil, T. An introduction to complex binary number system. In *2011 Fourth International Conference on Information and Computing* 229–232 (IEEE, 2011).
68. Zaini, H. & Deshmukh, R. G. A novel method for arithmetic operations using complex binary number system and the reconversion of the result to the decimal complex number system. In *IEEE SoutheastCon, 2003. Proceedings* 31–37 (IEEE, 2003).
69. Jamil, T. Complex binary associative dataflow processor—a tutorial. In *SoutheastCon 2018* 1–3 (IEEE, 2018).
70. Aoki, T., Ohki, Y. & Higuchi, T. Redundant complex number arithmetic for high-speed signal processing. In *VLSI Signal Processing, VIII* 523–532 (IEEE, 1995).
71. Ohi, Y., Aoki, T. & Higuchi, T. Redundant complex number systems. In *Proceedings 25th International Symposium on Multiple-Valued Logic* 14–19 (IEEE, 1995).
72. Ercegovac, M. D. & Muller, J.-M. Design of a complex divider. In *Advanced Signal Processing Algorithms, Architectures, and Implementations XIV* 51–59 (SPIE, 2004).
73. Priest, D. M. Efficient scaling for complex division. *ACM Trans. Math. Softw.* **30**, 389–401 (2004).
74. Agrawala, V. S. & Maharaja, J. S. S. B. K. T. Vedic mathematics: sixteen simple mathematical formulae from the Vedas (Motilal Banarsidass, 1988). ISBN-10: 8120801636, ISBN-13: 978–8120801639.
75. O'Connor, J. J. & Robertson, E. F. The Indian sulbasutras (MacTutor by the School of Mathematics and Statistics, University of St Andrews). https://mathshistory.st-andrews.ac.uk/HistTopics/Indian_sulbasutras/.
76. Vazquez, A., Antelo, E. & Montuschi, P. A radix-10 SRT divider based on alternative BCD codings. In *2007 25th International Conference on Computer Design* 280–287 (IEEE, 2007).
77. Richardson, S. E. Exploiting trivial and redundant computation. In *Proceedings of IEEE 11th Symposium on Computer Arithmetic* 220–227 (IEEE, 1993).

78. Product Specification. LogiCORE IP Divider Generator V4.0. Xilinx, Inc, DS819, 1–27 (2011).
79. Bruguera, J. D. Radix-64 floating-point divider. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)* 84–91 (IEEE, 2018).
80. Nannarelli, A. Performance/power space exploration for binary64 division units. *IEEE Trans. Comput.* **65**, 1671–1677 (2016).
81. Carter, T. M. & Robertson, J. E. Radix-16 signed-digit division. *IEEE Trans. Comput.* **39**, 1424–1433 (1990).
82. Rust, I. & Noll, T. G. A digit-set-interleaved radix-8 division/square root kernel for double-precision floating point. In *2010 International Symposium on System on Chip* 150–153 (IEEE, 2010).
83. Sharangpani, H. P. & Barton, M. L. *Statistical Analysis of Floating-Point Flaw in the PentiumTM Processo* (Intel Corporation, 1994).
84. Srinivas, H. R. & Parhi, K. K. A fast radix-4 division algorithm and its architecture. *IEEE Trans. Comput.* **44**, 826–831 (1995).

Acknowledgements

This project has received funding from the Estonian Research Council Institutional Research Projects IUT19-11, PUT1435, PRG780, EAS—Enterprise Estonia under project number: EU60351 and partly from the European Union's Horizon 2020 Research and Innovation Program under Grant 668995 and a preliminary patent is applied in Estonia based on the research work of developing a new state of the art USP-Awadhoot division algorithm. Application no- 70390, dated- June 2020. We acknowledge the support from Mr. Sunil M. Patankar, researcher & honorary lecturer at Kavi Kulguru Kalidas Sanskrit University, Ramtek, India, in finalizing the algorithm and supporting for the patent application.

Author contributions

U.S.P. conceived and led the research and contributed to all aspects of the project: project idea, algorithm design, circuit design and fabrication, testing, simulation and data analysis. A.K. advised on all parts of the project, patent and legal documentation. M.E.F. contributed to simulations, circuit design and testing. All authors discussed the results and contributed to the preparation of the manuscript.

Competing interests

The authors declare no competing interests.

Additional information

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1038/s41598-023-28343-3.

Correspondence and requests for materials should be addressed to U.S.P.

Reprints and permissions information is available at www.nature.com/reprints.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Appendix 5

Sample examples of the proposed novel USP-Awadhoot divider.

Example 1: - 28 ÷ 7 = 4

Step 1 Select Dividend (D_d) and Divisor (D_r)	D_d = 28 = 028 D_r = 7
Step 2 New divisor (ND_r) and Flag Digit (FD)	ND_r = 7 + 3 = 10 FD = +3
Step 3 Modified Divisor (MD_r) and Number of Zeros cancelled (NZC)	MD_r= 1 NZC= 1
Step 4 Group Dividend	Group3 = 0, Group2 = 2, Group1 = 8
Awadhoot Matrix processing	

Iteration 1

MD_r= 1 FD = +3		0		2		8	Group Dividend
	0						R_n
							Gross Dividend (G_rD_{dn})
							P-Term
							Net Dividend (ND_{dn})
							S-Term
							Group Quotient (GQ_n)

MD_r= 1 FD = +3		0		2		8	Group Dividend
	0		0				R_n
	0						Gross Dividend (G_rD_{dn})
	0						P-Term
	0						Net Dividend (ND_{dn})
	0						S-Term
	0						Group Quotient (GQ_n)

MD_r= 1 FD = +3		0		2		8	Group Dividend
	0		0				R_n
	0		2				Gross Dividend (G_rD_{dn})
	0		3*0 = 0				P-Term
	0						Net Dividend (ND_{dn})
	0						S-Term
	0						Group Quotient (GQ_n)

MD_r= 1 FD = +3		0		2		8	Group Dividend
	0		0				R_n
	0		2				Gross Dividend (G_rD_{dn})
	0		0				P-Term
	0		2				Net Dividend (ND_{dn})
	0		1*2 = 2				S-Term
	0						Group Quotient (GQ_n)

MD_r= 1 FD = +3		0		2		8	Group Dividend
	0		0		0		R_n
	0		2				Gross Dividend (G_rD_{dn})
	0		0				P-Term
	0		2				Net Dividend (ND_{dn})
	0		1*2 = 2				S-Term
	0		2				Group Quotient (GQ_n)

Iteration 2

MD_r= 1 FD = +3		0		2		8	Group Dividend
	0		0		0		R_n
	0		2		8		Gross Dividend (G_rD_{dn})
	0		0				P-Term
	0		2				Net Dividend (ND_{dn})
	0		1*2 = 2				S-Term
	0		2				Group Quotient (GQ_n)

MD_r= 1 FD = +3		0		2		8	Group Dividend
	0		0		0		R_n
	0		2		8		Gross Dividend (G_rD_{dn})
	0		0		6		P-Term
	0		2				Net Dividend (ND_{dn})
	0		1*2 = 2				S-Term
	0		2				Group Quotient (GQ_n)

MD_r= 1 FD = +3		0		2		8	Group Dividend
	0		0		0		R_n
	0		2		8		Gross Dividend (G_rD_{dn})
	0		0		6		P-Term
	0		2		14		Net Dividend (ND_{dn})
	0		1*2 = 2				S-Term
	0		2				Group Quotient (GQ_n)

In the last iteration net dividend (ND_d) must always be less than the divisor or zero. Last Net dividend (ND_{dn}) is 14, which is greater than Divisor (D_r). The Remainder is initialized to zero, i.e., (*R*) = R_{AQ} . The final R_{AQ} value is obtained during the calculation of the additional quotient (AQ), and Quotient (*Q*) = Partial Quotient (PQ_n) + Additional Quotient (AQ), where the Additional Quotient (AQ) is derived by initializing the count to zero and subtracting the Divisor (D_r) from the last iteration of Net Dividend (ND_d) and incrementing the count by one. We continue the same process until we get a subtraction result of zero or less than the divisor (D_r).

Count	1	2
Last Iteration Net dividend (LNDd) or copy Sub result for next subtraction	14	7
Divisor (Dr)	7	7
Subtraction result	14-7=7	7-7=0
Is Sub Result > Divisor (Dr)	7=7	0<7

Last count Sub result = 0 Count = 2

∴ Final Quotient = 2 + 2 = 4 and Remainder = 0

Answer - 28 / 7 = 4

Example 2: - 3657 ÷ 69 = 53

Step 1 Select Dividend (D_d) and Divisor (D_r)	D_d = 3657 = 03657 D_r = 69
Step 2 New divisor (ND_r) and Flag Digit (FD)	ND_r = 69 + 1 = 70 FD = +1
Step 3 Modified Divisor (MD_r) and Number of Zeros cancelled (NZC)	MD_r= 7 NZC= 1
Step 4 Group Dividend	Group5 = 0, Group4 = 3, group3 = 6, Group2 =5, Group1 = 7
Awadhoot Matrix processing	

Iteration 1

MD_r= 7 *FD* = +1		0		3		6		5		7
	0		0							
	0									
	(+1)*0=0									
	0									
	0									
	0									

Iteration 2

MD_r= 7 *FD* = +1		0		3 As MD_r>3		6		5		7
	0		0							
	0			So, the Quotient is zero, and the remainder is 3						
	(+1)*0=0									
	0									
	0									
	0									

MD_r= 7 *FD* = +1		0		3 As MD_r>3		6		5		7
	0		0							
	0			So, the Quotient is zero, and the remainder is 3						
	(+1)*0=0									
	0									
	0									
	0			0						

Iteration 3

MD_r = 7 *FD* = +1		0		3 As MD_r>3		6		5		7
	0		0		3					
	0			So, the Quotient is zero, and the remainder is 3						
	(+1)*0=0									
	0									
	0									
	0			0						

MD_r = 7 *FD* = +1		0		3 As MD_r>3		6		5		7
	0		0		3					
	0			So, the Quotient is zero, and the remainder is 3	36					
	(+1)*0=0									
	0									
	0									
	0			0						

MD_r = 7 *FD* = +1		0		3 As MD_r>3		6		5		7
	0		0		3					
	0			So, the Quotient is zero, and the remainder is 3	36					
	(+1)*0=0				(+1)*0=0					
	0									
	0									
	0			0						

MD_r = 7 *FD* = +1		0		3 As MD_r>3		6		5		7
	0		0		3					
	0			So, the Quotient is zero, and the remainder is 3	36					
	(+1)*0=0				(+1)*0=0					
	0				36					
	0				7*5=35					
	0			0						

Iteration 4

MD_r= 7 *FD* = +1		0		3 As MD_r>3		6		5		7
	0		0		3		1			
	0			So, the Quotient is zero, and the remainder is 3	36		15			
	(+1)*0=0				(+1)*0=0		(+1)*5=5			
	0				36		20			
	0				7*5=35		7*2=14			
	0			0	5					

MD_r= 7 *FD* = +1		0		3 As MD_r>3		6		5		7
	0		0		3		1		6	
	0			So, the Quotient is zero, and the remainder is 3	36		15			
	(+1)*0=0				(+1)*0=0		(+1)*5=5			
	0				36		20			
	0				7*5=35		7*2=14			
	0			0	5		2			

Iteration 5

MD_r= 7 *FD* = +1		0		3 As MD_r>3		6		5		7
	0		0		3		1		6	
	0			So, the Quotient is zero, and the remainder is 3	36		15		67	
	(+1)*0=0				(+1)*0=0		(+1)*5=5			
	0				36		20			
	0				7*5=35		7*2=14			
	0			0	5		2			

MD_r = 7 *FD* = +1		0		3 As MD_r>3		6		5		7
	0		0		3		1		6	
	0			So, the Quotient is zero, and the remainder is 3	36		15		67	
	(+1)*0=0				(+1)*0=0		(+1)*5=5		(+1)*2=2	
	0				36		20			
	0				7*5=35		7*2=14			
	0			0	5		2			

MD_r = 7 *FD* = +1		0		3 As MD_r>3		6		5		7
	0		0		3		1		6	
	0			So, the Quotient is zero, and the remainder is 3	36		15		67	
	(+1) * 0=0				(+1) * 0=0		(+1) * 5=5		(+1) * 2=2	
	0				36		20		69	
	0				7*5=35		7*2=14			
	0			0	5		2			

Number in Quotient Row – 052 = 52 and
Net dividend = 69, which is equal to the divisor, so we add 1 to the quotient.
Final Quotient = 52+1 = 53 and remainder = 0
Answer - 3657 ÷ 69 = 53

Appendix 6

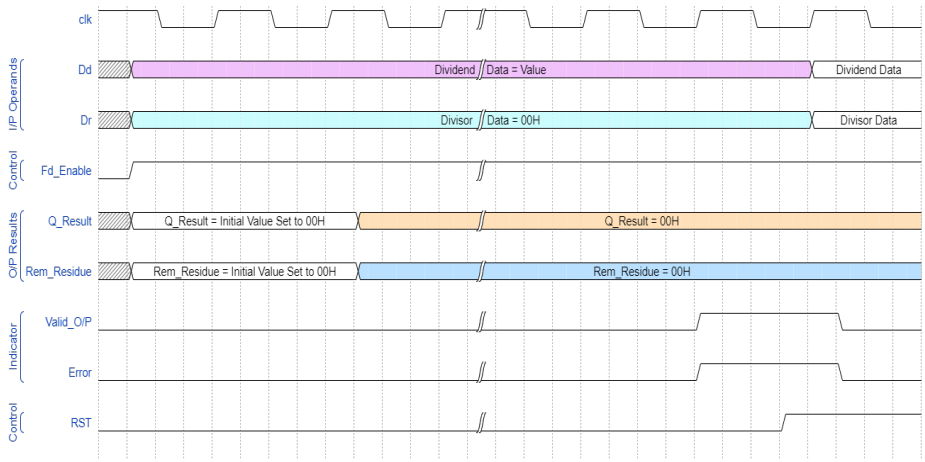
Sample example of the proposed complex division by the Baudhayan-Pythagorean Triplet Method.

Example 3: - (1+18*i*) ÷ (3+4*i*) = (3+2*i*)

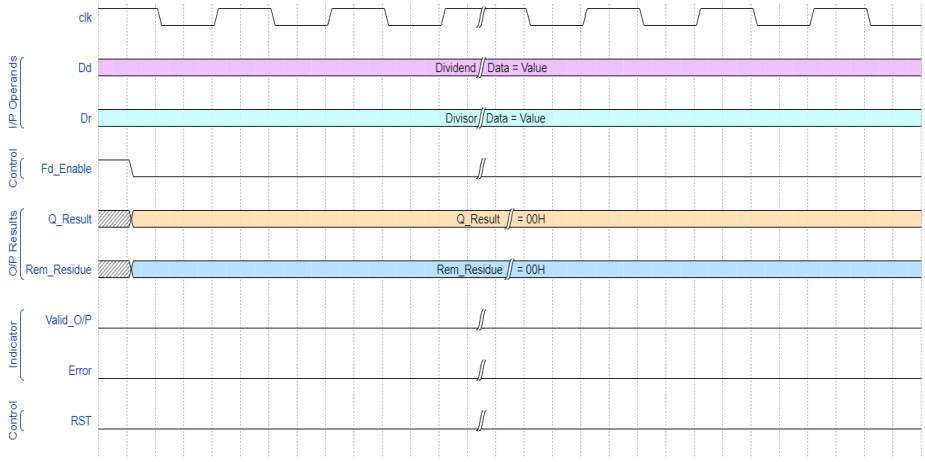
Step 1 Select Dividend (C_D_d) and Divisor (C_D_r)	$C\_D_d = x_1 + y_1 i$ = 1+18*i* $C\_D_r = x_2 + y_2 i$ = 3+4*i*
Step 2 Cartesian coordinates of Dividend (C_D_d) and Divisor (C_D_r) $xD_d = x_1$, $yD_d = y_1$, $xD_r = x_2$ and $yD_r = y_2$	$xD_d = x_1 = 1$ $yD_d = y_1 = 18$ $xD_r = x_2 = 3$ $yD_r = y_2 = 4$
Step 3 Derive the triplet product term (TP_Term) value. TP_Term1 = $(xD_d \times xD_r)$ TP_Term2 = $(yD_d \times yD_r)$ TP_Term3 = $(xD_r \times yD_d)$ TP_Term4 = $(xD_d \times yD_r)$	TP_Term1 = $(xD_d \times xD_r)$ = (1X3) = 3 TP_Term2 = $(yD_d \times yD_r)$ = (18X4) = 72 TP_Term3 = $(xD_r \times yD_d)$ = (3X18) = 54 TP_Term4 = $(xD_d \times yD_r)$ = (1X4) = 4
Step 4 Derive the triplet term (T_Term) value, where T_Term = $(xD_r)^2 + (yD_r)^2$	T_Term = $3^2 + 4^2$ = 25
Step 5 Derive the triplet matrix term (Mat_Term) value. Mat_Term1 = TP_Term1 + TP_Term2 Mat_Term2 = TP_Term3 - TP_Term4	Mat_Term1 = 3 + 72 = 75 Mat_Term2 = 54 – 4 = 50
Step 6 Provide Mat_Term1, Mat_Term2, and T_Term values to the USP-Awadhoot divider for the final division sub-process.	
Step 7 USP-Awadhoot divider-1 C_D_{d1} = Mat_Term1 = 75 C_D_{r1} = T_Term = 25 Perform the division as explained by the USP-Awadhoot divider 1 and get the value for the result C_Q_r and C_Rem_r. After processing USP-Awadhoot divider 1, we get, C_Q_r = 3 C_Rem_r = 0	Step 8 USP-Awadhoot divider-2 C_D_{d2} = Mat_Term2 = 50 C_D_{r2} = T_Term = 25 Perform the division as explained by the USP-Awadhoot divider 2 and get the value for the result C_Q_i and C_Rem_i. After processing USP-Awadhoot divider 1, we get, C_Q_i = 2 C_Rem_i = 0
Step 9 Concatenate the computational results from step 7 and step 8 to restructure Cartesian coordinates, to get the final quotient and remainder of a complex division. Final Quotient = C_Q = ($C\_Q_r + C\_Q_i$) = (3 + 2*i*) Final Remainder = C_Rem = ($C\_Rem_r + C\_Rem_i$) = (0 + 0*i*)	

Appendix 7

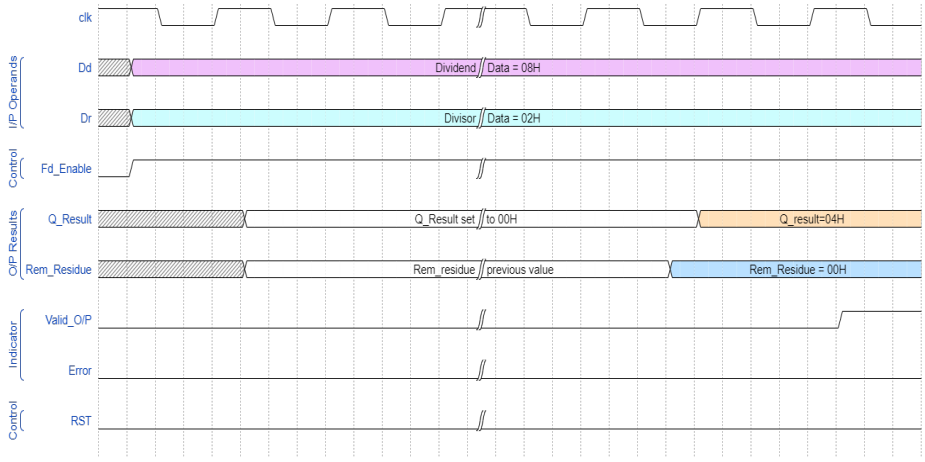
USP-Awadhoot divider functional waveforms.



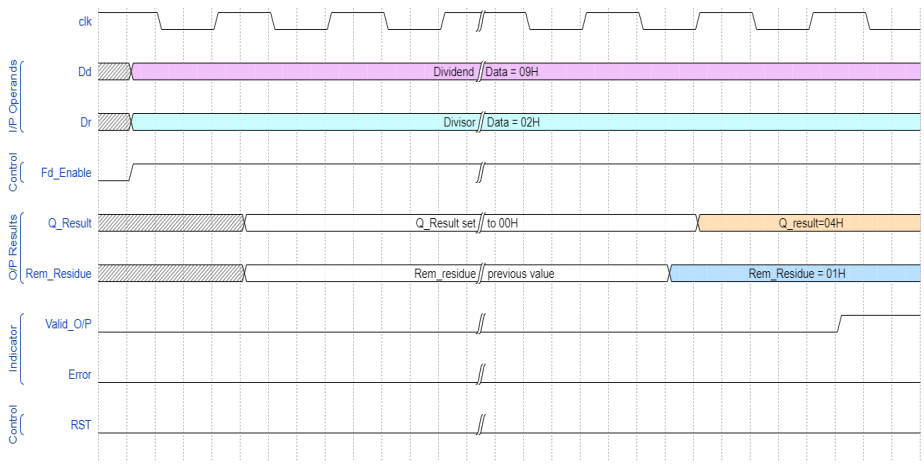
Waveform for divide by zero working condition.



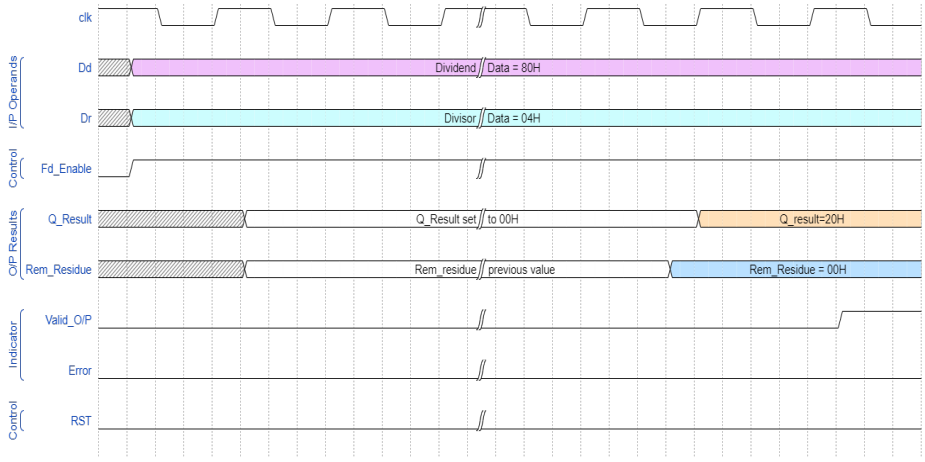
Waveform for inactive F_{d}_enable working condition.



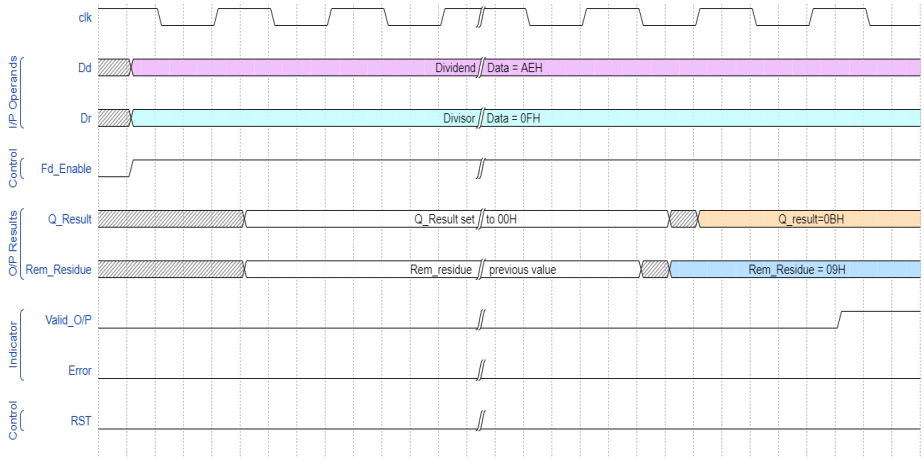
Waveform for a 4-bit operand, resulting in zero remainder.



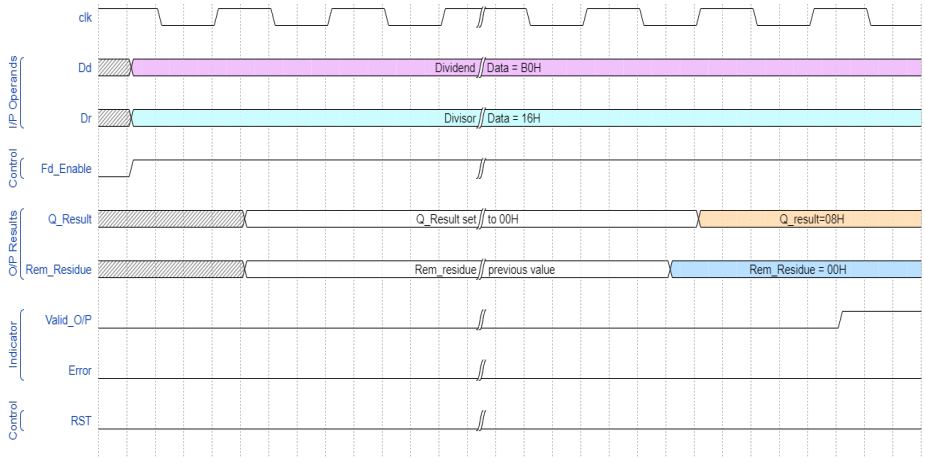
Waveform for a 4-bit operand, resulting in a non-zero remainder.



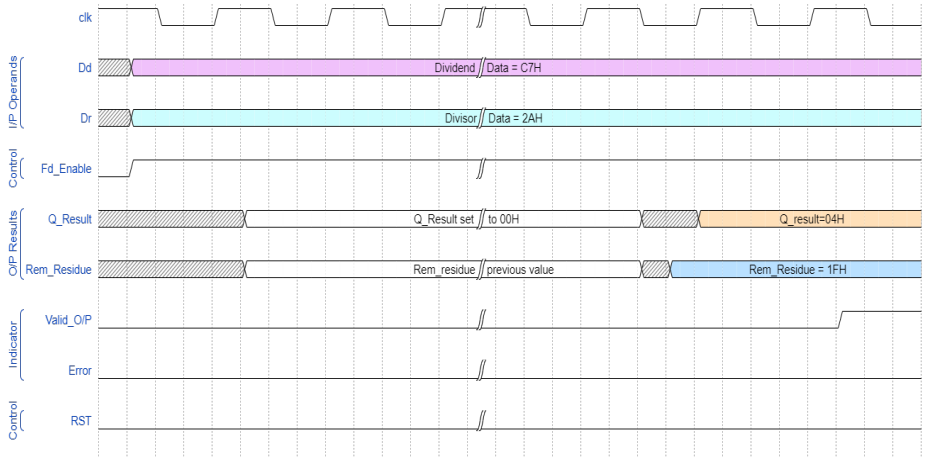
Waveform for an 8-bit dividend and 4-bit divisor, resulting in zero remainder.



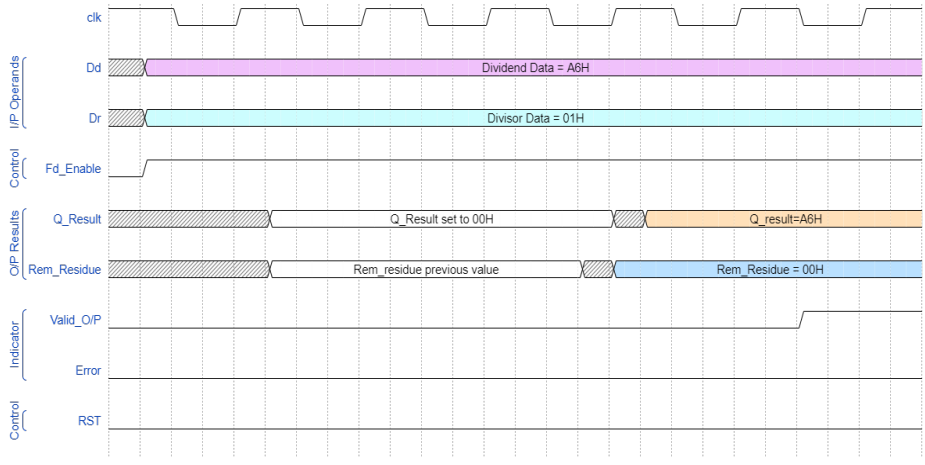
Waveform for an 8-bit dividend and 4-bit divisor, resulting in a non-zero remainder.



Waveform for an 8-bit operand, resulting in zero remainder.



Waveform for an 8-bit operand, resulting in a non-zero remainder.



Waveform for divider by unity.

Curriculum Vitae

1 Personal Data

Name	Udayan Sunil Patankar
Citizenship	India
Language Competence	English, Hindi, Marathi

2 Contact Information

Address	Energia 5-17, Tallinn, Estonia, 13415.
Contact number	+372-58791581
E-mail	Udayan.patankar45@gmail.com

3 Education

2017–2025	Tallinn University of Technology Thomas Johann Seebeck Department of Electronics Ph.D. Studies
2012–2014	G.H.Raisoni College of Engineering, Nagpur Affiliated to RTM Nagpur University, India M. Tech- Electronics
2008–2011	Shree Ramdeobaba Kamla Nehru Engineering College Affiliated to RTM Nagpur University, India B.E. in Electronic Design Technology

4 Professional Experience

2017–2021	Tallinn University of Technology, Estonia, Early Stage Researcher
2015–2017	Shri Ramdeobaba College of Engineering and Management, India, Assistant Professor
2014–2014	Brick and Byte Innovative Products Limited Mumbai, India, Hardware Design Engineer
2013–2014	Linyi InfoTop Network, Linyi CHINA (Industrial Project) Embedded Hardware Design Engineering
2012–2013	SM Wireless Solutions, Pvt. Ltd, India, Trainee Analog Layout Engineer
2011–2012	Ambuja Cements Limited, India, Senior Executive

5 Professional Training

05/2010–06/2021	Trainee, Airport Authority of India(AOI), Mumbai, India
06/2018	Short course on "IC Processing Technology," at College of Engineering, Rochester Institute of Technology, Rochester, USA
02/2019–03/2019	Winter school "Chip fab of the future!" Infineon Technologies Austria AG

6 Awards/Achievements

2010	IIT KHARAGPUR finalist in Project Competition
2010	POLARIS - Project Competition, 3rd prize
2009	National Level Circuit Design Competition (NYSS), 1st prize
2009	OPEN HARDWARE Techno Vision, SRKNEC, 1st prize.
1998–1999	1st prize in the International Karate Championship, Nepal
1996	Gold Medal, 4th National Martial Arts championship organized by Martial Arts Association of India

7 Expert lectures

2015	Delivered Expert Lecture on "Energy Efficient Technology" MSME Gov. Of India on July 24, 2015

Elulookirjeldus

1 Isikuandmed

Nimi	Udayan Sunil Patankar
Kodakondsus	India
Keelteoskus	Inglise keel, Hindi keel, Marathi keel

2 Kontaktandmed

Address	Energia 5-17, Tallinn, Eesti, 13415.
Contact number	+372-58791581
E-post	Udayan.patankar45@gmail.com

3 Haridus

2017–2025	Tallinna Tehnikaülikool Thomas Johann Seebecki elektroonikainstituut doktoriõpe
2012–2014	G.H.Raisoni College of Engineering, Nagpur Affiliated to RTM Nagpur University, India M. Tech- Electronics
2008–2011	Shree Ramdeobaba Kamla Nehru Engineering College Affiliated to RTM Nagpur University, India B.E. in Electronic Design Technology

4 Teenistuskäik

2017–2021	Tallinna Tehnikaülikool, Eesti, nooremteadur
2015–2017	Shri Ramdeobaba College of Engineering and Management, India, Assistant Professor
2014–2014	Brick and Byte Innovative Products Limited Mumbai, India, Hardware Design Engineer
2013–2014	Linyi InfoTop Network, Linyi CHINA (Industrial Project) Embedded Hardware Design Engineering
2012–2013	SM Wireless Solutions, Pvt. Ltd, India, Trainee Analog Layout Engineer
2011–2012	Ambuja Cements Limited, India, Senior Executive

5 Täiendkoolitus

05/2010–06/2021	Trainee, Airport Authority of India(AOI), Mumbai, India
06/2018	Short course on “IC Processing Technology,” at College of Engineering, Rochester Institute of Technology, Rochester, USA
02/2019–03/2019	Winter school “Chip fab of the future!” Infineon Technologies Austria AG

6 Tunnustused

2010	IIT KHARAGPUR finalist in Project Competition
2010	POLARIS - Project Competition, 3rd prize
2009	National Level Circuit Design Competition (NYSS), 1st prize
2009	OPEN HARDWARE Techno Vision, SRKNEC, 1st prize
1998–1999	1st prize in the International Karate Championship, Nepal
1996	Gold Medal, 4th National Martial Arts championship organized by Martial Arts Association of India

7 Erialaloengud

2015	Delivered Expert Lecture on “Energy Efficient Technology” MSME Gov. Of India on July 24, 2015.

ISSN 2585-6901 (PDF)