Applied and Computational Mathematics
2024; 13(4): 83-93
http://www.sciencepublishinggroup.com/j/acm
doi: 10.11648/j.acm.20241304.12
ISSN: 2328-5605 (Print); ISSN: 2328-5613 (Online)

Novel Integer Division for Embedded Systems: Generic Algorithm Optimal for Large Divisors

Mervat Mohamed Adel Mahmoud*, Nahla Elazab Elashker

Microelectronics Department, Electronics Research Institute, Cairo, Egypt

Email address:

mervat-m@eri.sci.eg (Mervat Mohamed Adel Mahmoud), nahlaelazab@eri.sci.eg (Nahla Elazab Elashker)
*Corresponding author

To cite this article:

Mervat Mohamed Adel Mahmoud, Nahla Elazab Elashker. Novel Integer Division for Embedded Systems: Generic Algorithm Optimal for Large Divisors. *Applied and Computational Mathematics, 13*(4), 83-93. https://doi.org/10.11648/j.acm.20241304.12

Received: 9 June 2024; **Accepted:** 1 July 2024; **Published:** 24 July 2024

Abstract: The integer Constant Division (ICD) is the type of integer division in which the divisor is known in advance, enabling pre-computing operations to be included. Therefore, it can be more efficient regarding computing resources and time. However, most ICD techniques are restricted by a few values or narrow boundaries for the divisor. On the other hand, the main approaches of the division algorithms, where the divisor is variable, are digit-by-digit and convergence methods. The first techniques are simple and have less sophisticated conversion logic for the quotient but also have the problem of taking significantly long latency. On the contrary, the convergence techniques rely on multiplication rather than subtraction. They estimate the quotient of division providing the quotient with minimal latency at the expense of precision. This article suggests a precise, generic, and novel integer division algorithm based on sequential recursion with fewer iterations. The suggested methodology relies on extracting the division results for non-powers-of-two divisors from those for the closest power-of-two divisors, which are obtained simply using the right bit shifting. To the authors' best knowledge of the state-of-the-art, the number of iterations in the recurrent variable division is half the divisor bit size, and the Sweeney, Robertson, and Tocher (SRT) division, which is named after its developers, involves log_2(n) iterations. The suggested algorithm has an [(m/(n-1))-1] number of recursive iterations, where m and n are the number of bits of the dividend and the divisor, respectively. The design is simulated in the Vivado tool for validation and implemented with a Zynq UltraScale FPGA. The technique performance depends on the number of nested divisions and the size of a LUT. The two factors change according to the value of the divisor. Nevertheless, the size of the LUT is proportional to the range and the number of bits of the divisor. Furthermore, the equation that controls the number of nested blocks is illustrated in the manuscript. The proposed technique applies to both constant and variable divisors with a compact hardware area in the case of constant division. The hardware implementation of constant division has unlimited values for dividends and divisors with a compact hardware area in the case of large divisors. However, using the design in the hardware implementation of variable division is up to 64-bit dividend and 12-bit divisor. The result analysis demonstrates that this algorithm is more efficient for constant division for large numbers.

Keywords: Algorithms, Arithmetic Functions, Embedded Systems, Hardware Implementation, Integer Division

1. Introduction

Recently, the division process has been manipulated and required for many operations that are needed for significant applications such as the normalization known as Batch-Norm, which is frequently used in deep neural network training. This normalization technique is a crucial component of neural network architectures and can increase both convergence and generalization in most applications [1]. In addition, it supports the current trend of building artificial neural networks directly in hardware, which enables significant advances in network size scaling and gets beyond the Von-Neuman bottleneck constraint [2]. Other examples of the applications that are being emphasised are the digital signature algorithm (DSA), which uses division in both the signing and signature verification algorithms [3], the ElGamal cryptosystem that utilises a specific prime of 21279 only as a divider in the

decryption process [4], and the Rivest-Shamir-Adleman (RSA) cryptosystem, in which the secret key is generated by dividing by the public key [5].

A high-performance implementation of the integer division operation is the most challenging of the four basic arithmetic operations (i.e., addition, subtraction, multiplication, and division). As it is generally slower than the other three basic arithmetic operations if performed using general-purpose computer architecture [6], and most costly that takes up more space in the case of Very Large Scale Integration (VLSI) circuitry [7]. On the other side, it is used extensively in various applications that necessitate the creation of a dedicated operator in embedded systems design defined as a "divider by a constant"[8]. Moreover, the performance of many applications such as Digital Signal Processing (DSP) applications [9], cryptography [10], and deep learning [2] are highly affected by the quality of the used division algorithm. Particularly, when VLSI implements more advanced DSP algorithms, the divider becomes an essential component for digital architecture [11]. Simply because calculating the ratio of two values is necessary for many signal-processing methods.

Initially, the division algorithms can be classified according to whether the divider is a predetermined constant [7] or a variable [12–15]. Specifically, the Integer Constant Division (ICD) is the type of integer division in which the divisor is known in advance, enabling pre-computing operations to be included. The core trick behind all ICDs methods is that every operation with the constant argument(s) can be made more efficient in terms of computing resources and/or time, when comparing an ICD circuit implementation with variable parameters one [16]. However, most of the proposed ICD techniques in literature are restricted by a few examples or very narrow boundaries [6].

On the other hand, the main approaches used to depict division algorithms, where the divisor is variable, are the digit-by-digit [12] and convergence methods [13]. The digit-by-digit techniques are similar to the simple division technique used with paper and pencil, where the results are computed digit by digit, with the most significant digit first [17]. They have the feature of being simple and have less sophisticated conversion logic for the quotient but also have the problem of taking significantly long latency [18, 19]. On the contrary, the convergence techniques rely on multiplication rather than subtraction. They estimate the quotient of division. It computes more than one digit of the quotient in a single iteration to minimize the number of iterations. It provides the quotient with minimal latency at the expense of precision. Therefore, additional iterations are needed to update the approximated result to a more correct one. Originally, four prime algorithms followed this concept which are Newton-Raphson Algorithm (NRA) [20, 21], Gold-Schmidt Algorithm (GSA)[7], Series Expansion Algorithm (SEA) [22], and the Taylor Series Expansion Algorithm (TSEA) [23].

Basically, many algorithms were presented for the general-purpose computer architecture, as a result, the optimization of division by a compile-time constant integer is well documented [24, 25]. Besides, analog and digital circuits have been used to implement these techniques without using a general-purpose Central Processing Unit (CPU) architecture. According to some research, the division techniques are meant to be used as co-processors in an application-specific instruction processor (ASIP) [26], as part of a System On Chip (SoC) [23], or as an Application-Specific Integrated Circuit (ASIC) [27]. The criticality of the application, such as whether it is time-critical or space-critical, generally determines the selection of the optimal divider option and the optimal implementation scheme. On the basis of this, one must choose the ideal substitute for the divider circuit or the block being implemented.

Generally, all techniques try to carry out the division in different ways targeting the mathematical accuracy, maximum operating frequency, and minimum number of clock cycles. Marching towards these targets, a generic algorithm for constant division is suggested in this paper. This manuscript is organized in the following manner: A literature review of recent hardware integer division is presented in Section 2. The algorithm is explained in detail in Section 3. Furthermore, the algorithm performance and hardware implementation quality are measured and compared with state-of-the-art in Section 4. Finally, concluding remarks are given in Section 5.

2. Hardware Implementation of Integer Division in Literature

Three general techniques are the main for hardware implementation of m-bit dividend by n-bit divider, a/b operation. Firstly, the binary long-division technique, adopting the Euclidean division approach, has $(m - n)$ number of successive multiplications. It starts by comparing the n most significant bits (MSBs) of the dividend with the divisor, and then at each cycle, the dividend is shifted one bit. It provides one quotient digit per iteration, which results in a large rise in computation steps and, in some cases, imposes severe performance constraints on the system [17]. Ugurdag et al. in 2017 [17] addressed the problem of finding the quotient and remainder of the Euclidean division of an unsigned integer by a small constant integer such as 3 or 10. The suggested solutions in this research work well with an FPGA's hardware resources as they use compact LUT. The authors stated that while the temporal overhead is quite minor compared to the area overhead, both are independent of the divisor. Furthermore, the article investigates whether using two sequential dividers or a single atomic divider is preferable when performing division by the product of two constants. It demonstrated that the composite dividers are smaller and consistently faster on FPGAs. The authors explained this result because the critical path of the composite architecture includes fewer lookup tables. They also consider the case that only the quotient or remainder is required.

Secondly, another well-known method is the addition-subtraction or add-shift method. It is performed using successive additions of the divisor till it reaches the dividend

or successive subtraction of the divisor from the dividend. The number of successive addition/subtraction operations is equal to the output quotient, so it can be said that the number of successive addition/subtraction operations needed is up to $(2^{m-n}-1)$. Yassin et al. in 2015 [16] presented a hardware implementation for the constant division using the add-shift technique and set the condition of add-shift schemes that had been modified from Warren algorithm [28] to eliminate the integer multiplication and to round the unsigned result to the nearest integer. However, his technique is bounded to specific numerator bit width and limited to certain constant dividers. In more detail, it is applicable only for numerator bit width up to 13 bit for 5, 6, 7, and 9 dividers while 16 bit in case of divider equals 3, and has limited divider values. In the same context, many published methods are limited to specific dividers values [16]. On the whole, both aforementioned methods require a lot of power and considerable delay.

Thirdly, from the literature, many approaches attempted to decrease the complexity of the integer division by generating an approximate value of $(1/b)$ and then multiplying it by a. Nigel Jones, in 2009 [29], pre-calculated $(1/B)$ using a software calculator. Then he calculates the quotient using one of the following two equations where A and M are of 32-bit

$$Q = ((A * M) >> 16) >> S \tag{1}$$

$$Q = (((A * M) >> 16) + A) >> 1 >> S \tag{2}$$

Alternatively, replacing the division of a by b with the division of $c*a$ or $(c*a+c)$ by m for convenient integers c and m chosen so that they approximate the reciprocal: c/m to $1/b$. These methods are extremely beneficial when m is chosen as a power of two and b is a constant so that c and m can be pre-calculated. Meanwhile, the literature contains several bounds on the distance between c/m and the divisor b. Some of these bounds are tight, while others are not [25].

Regarding the hardware implementation of the multiplication by reciprocal division strategy, Deshpande et al., in 2021, [30] provides two parameterizable FPGA-based hardware designs for computing the multiplicative inverse using the previously published fast constant-time Greatest Common Divisor (GCD) algorithm [31] to accelerate the ElGamal cryptosystem [4]. The two design architectures, a full-width design, and a sequential design, target different applications. Since the first design offers a quick version that requires more resources so it does not fit many tested FPGA chips. While the second design offers a somewhat slower version that is more space-efficient and has a superior time-area product. Although the constant-time feature of these designs protects against timing-based attacks, the authors proposed a modular inverse targeting a specific prime of 21279 only as a divider, which is required for ElGamal decryption, and they assumed that the multiplications and divisions are by powers of 2. The hardware is implemented in VHDL with Xilinx Vivado 2019.2 for Zynq UltraScale +XCZU7EG.

The suggested methodology in this paper is not based on the convergence method as it satisfies an accurate result. Although our algorithm belongs to the sequential recurrences, which usually have $(m-n)$ number of iterations, the suggested algorithm has around m/n number of iterations, as demonstrated in the next section. This strategy is not restricted to a small positive constant. It is valid for a wide range of operands. However, other methods designed for hardware implementation generally do not scale to integers with thousands or millions of decimal digits. These frequently occur in embedded systems applications, for example, in modular reductions in cryptography [32]. Thus, a generic solution is offered and compared to the current state of the art.

3. Methodology

The suggested methodology relies on extracting the division results for non-powers-of-two divisors from those for the closest power-of-two divisors, which are obtained simply using the right bit shifting. This technique generates an integer quotient and a non-negative integer remainder as outputs of the division procedure. In the binary number system, each bit has a weight that equals a product of two multiplied by the weight of its right-hand side counterpart, so shift-right (SR) a binary number by 1-bit corresponds to division by two. For more bit-shifting, shift-right by 2-bit, 3-bit, and k-bit corresponds to division by 4, 8, and 2^k, respectively. And for numbers with non-zero least significant bits (LSBs), shift right outputs the quotient, Q, and the shifted LSBs are the remainder, R.

3.1. Generic Algorithm for Integer Division

The idea of the suggested algorithm is based on the simplicity of dividing by multiples of two. When the divisor is multiple of two, 2^k, the division operation is just right-shifting by k-bits. The divisors that can be represented as multiples of two, 2^k, will be called in this paper key numbers. For integer divisors between key numbers, the division is computed reliant on the division by the nearest smaller key to the divisor. For example, for the divisors in the range [5-7], [9-15], and [17-31], the key is 4, 8, and 16, respectively, and the same for any other divisor. For m-bit dividend A and n-bit divisor B, The division operation can be written as

$$A/B = Q + R/B \tag{3}$$

with a maximum value of R being $(B-1)$. The suggested algorithm's concept is based on utilizing nested division operations that divide by the key number of the divisor. The $block$ has two inputs, dividend and divisor, and two outputs, quotient, and remainder. Figure 1 shows the basic block diagram of the suggested nested division methodology. The dividend A is divided by the key, the nearest smaller number to the divisor that can be divided by 2. The dividend A is bit-shifted to produce the quotient Q_{sh} and remainder R_{sh}. Then, Q_{sh} is divided by the divisor B, which produces quotient Q_C and remainder R_C. The subscript letter sh is the abbreviation of shifting, while the subscript letter C is for computation.

The values of Q_c and R_c get from nested division blocks by recursive. They are calculated backward, from the last $block$ to the first one. This operation depends on the fact that the last $block$ has Qi_{sh} less than or equal to the divisor B, so its Qi_c and Ri_c are known. The i symbol in Qi_{sh}, Qi_c, and Ri_c subscripts refers to the order of the nested $block$. Then the final outputs of the $block$ are calculated from Q_{sh}, R_{sh}, Q_C, and R_C inside the $Result\ computation$ sub-block. The design of the $resultcomputation$ sub-block will be discussed later, though it has one multiplication, two addition/subtraction operations, a comparator, and access to a lookup table.

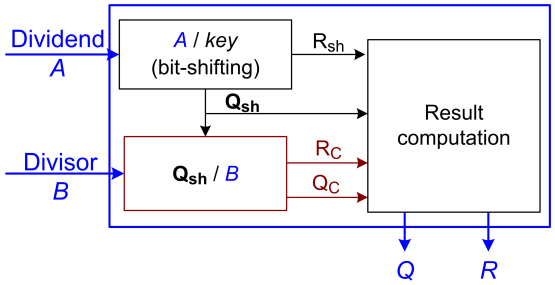

Figure 1. *The basic block diagram for the suggested nested division methodology.*

The Q_{sh}/B sub-block is considered as a new division by the same divisor B, so the same divisor design is used for it. The same for all nested Qi_{sh}/B sub-blocks till Qi_{sh} is smaller or equal to the divisor. Also, i in the subscript of the Qi_{sh}/B sub-block refers to the order of the nested division. Nevertheless, the LUT is generated for one time in the case of one constant divisor or set of divisors, and all nested operations use it. The structure of the suggested algorithm consists of N nested divisors, where $N = [m/(n-1)] - 1$. Figure 2 shows the block diagram of the divider with two nested divisions for $m/(n\text{-}1) = 3$. In other words, the number of dividend bits is around three times the number of bits of the divisor, m=3(n-1).

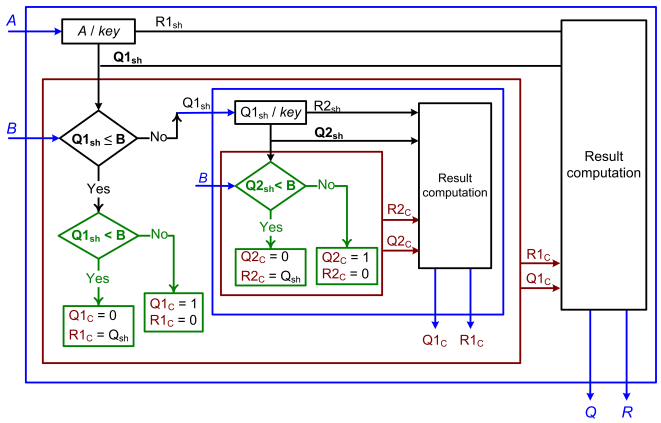

Figure 2. *block diagram of the divider with two nested divisions.*

For more clarification, Figure 3 shows the generic block diagram for the division methodology from the linked-list viewpoint.

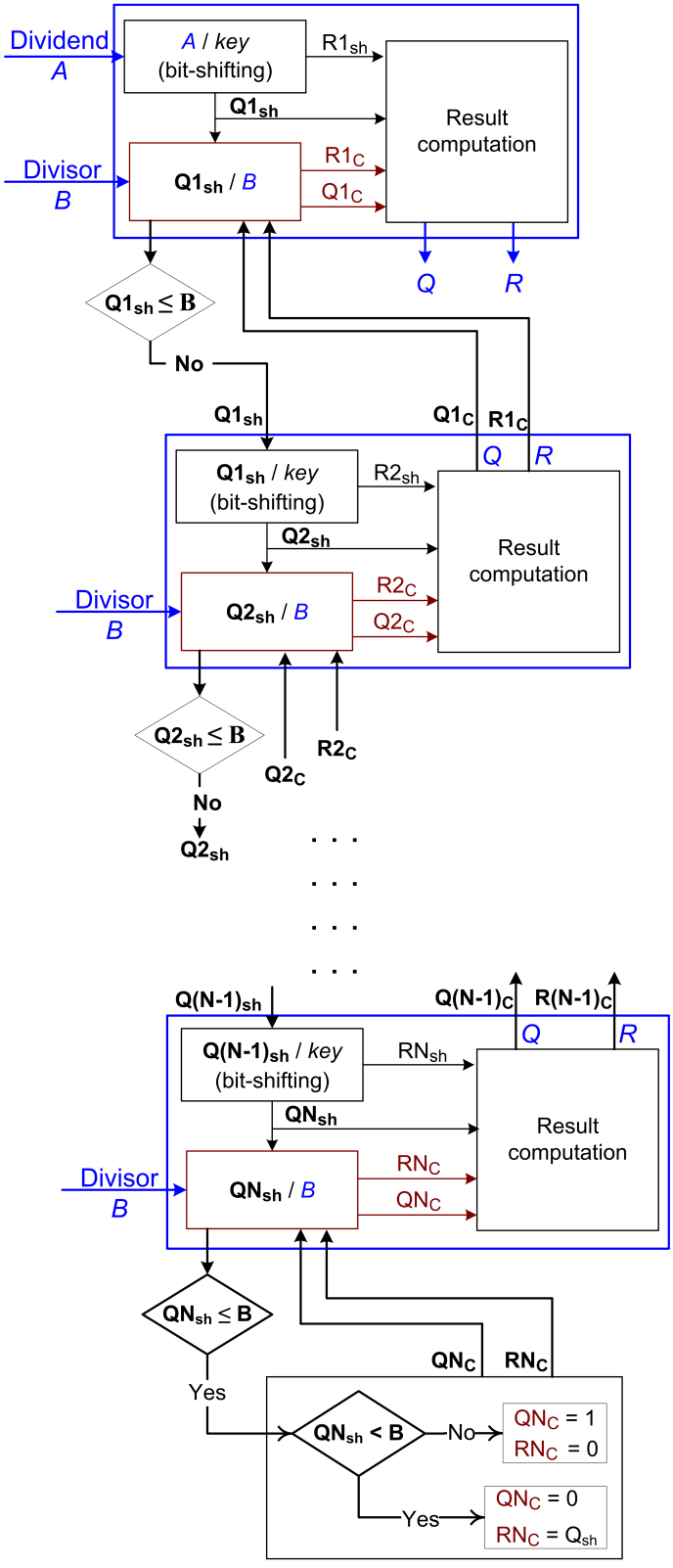

Figure 3. *Generic block diagram for the divider nested blocks.*

3.2. Algorithm Proof

Given an m-bit dividend $A = a_{m-1}.....a_1a_0$ and an n-bit divisor $B = b_{n-1}.....b_1b_0$. The quotient has (m-n+1) bits,

$Q = q_{m-n}.....q_1q_0$, and the remainder has n bits at its most extreme, $R = r_{n-1}.....r_1r_0$, where the maximum value of R is $B-1$. The target is the values of Q, and R. Eq. 3 can be represented as

$$A = Q * B + R \tag{4}$$

Moreover, when A is divided by $2^{(n-1)}$, which is named key, the quotient Q_{sh} and the remainder $R1_{sh}$ are produced, where

$$A = Q_{sh} * key + R_{sh} \tag{5}$$

And the value $B - key$ will be defined as x

$$B - key = x \tag{6}$$

The values of key, Q_{sh}, and $R1_{sh}$ are simply obtained. The key has the most significant bit equal to '1', and all other bits have a '0' value. By scanning the divisor from its MSB to LSB, the position of the first bit with a value equal to '1' is the position of the key bit with a '1' value. The Q_{sh}, and $R1_{sh}$ are parts from the dividend A. As all the parameters of Eq. 5 are available, we try to reach Eq. 4 from Eq. 5 to get Q and R.

$$\begin{aligned} A &= Q_{sh} * key + R_{sh} \\ &= Q_{sh} * (B - x) + R_{sh} \\ &= Q_{sh} * B - Q_{sh} * x + R_{sh} \\ &= (Q_{sh} - \frac{Q_{sh} * x}{B}) * B + R_{sh} \end{aligned} \tag{7}$$

As assumed before, when dividing, Q_{sh}, by the divisor B, the result quotient and remainder are Q_c and R_c respectively. The values of Q_c and R_c are known from the nested division. Thus

$$\because Q_{sh} = Q_c * B + R_c \qquad \therefore \frac{Q_{sh}}{B} = Q_c + \frac{R_c}{B} \tag{8}$$

So,

$$\begin{aligned} A &= (Q_{sh} - Q_c * x - \frac{R_c}{B} * x) * B + R_{sh} \\ &= (Q_{sh} - Q_c * x) * B - R_c * x + R_{sh} \end{aligned} \tag{9}$$

We will split the value of Q into two parts Q_a and Q_b, as shown in Eq. 10

$$A = (Q_a - Q_b) * B + R \tag{10}$$

Comparing Eq. 9 with Eq. 10, and defining Q_a as in Eq. 11

$$Q_a = Q_{sh} - Q_c * x \tag{11}$$

$$\therefore -Q_b * B + R = -R_c * x + R_{sh} \tag{12}$$

Now, the value of Q_a is calculated from the obtained Q_{sh}, Q_c, and x. While the values of Q_b, and R are got using a LUT that solves Eq. 12. In Eq. 12, the values of R_{sh}, R_c, and x are known, while the values of R, and Q_b are unknowns. Fortunately, R_{sh}, R_c, and x have relatively small values as x is smaller than the divisor and remainders. The size of LUT depends on the number of bits of the divisor and is independent on the dividend size so that we can use the same LUT for high dividend values. Nonetheless, high dividend values affect the number of nested operations N, not the size of the LUT.

3.3. Pseudo Code for m-bit Dividend and n-bit Divisor

```
PROCEDURE DIV(A, B) RETURN Q,R
    Qsh ← SHIFTRIGHT(A, n)
    Rsh ← A[0 : n − 1]
    IF Qsh ≤ B THEN
        IF Qsh ≤ B THEN
            Qc ← 0
            Rc ← Qsh
        ELSE
            Qc ← 1
            Rc ← 0
        END IF
    ELSE
        Qc, Rc = DIV(Qsh,B)
    END IF
    Qa ← Qsh − Qc × (B − 2^(n−1))
    LUT :   input:B, Rc   output:e, g
    IF g + Rsh ≤ B THEN
        Q ← Qa − e
        R ← g + Rsh
    ELSE
        Q ← Qa − e + 1
        R ← g + Rsh − B
    END IF
    RETURN Q,R
END PROCEDURE
```

3.4. Algorithm Procedure Steps for Each Nested Division

The division operation is rewritten as

$$A = (Q_a - Q_b) * B + R, \tag{13}$$

assuming that the values of Q_{sh}, R_{sh}, Q_c, and R_c are known for each nested division. Then the procedure is required to calculate the values of Q_a, Q_b, and R. Hence, three operations are used:

Operation 1: Multiply-Add operation

According to Eq.9, we will get $Q_{sh} - Q_c * x$ and name it Q_a. So Eq. 9 can be written as

$$Q_a = Q_{sh} - Q_c * x \tag{14}$$

Operation 2: Look-Up Table

To reduce the complexity of the LUT, we introduce two more variables, e and g, and rewrite Eq. 12 as

$$\begin{aligned} R_c * (B - key) &= Q_b * B + R_{sh} - R \\ &= e * B - g \end{aligned} \tag{15}$$

The LUT is designed according to the value of the divisor, with all the corresponding possible R_c values, where R_c is always less than the divisor. An example of the LUT for a constant divisor of 13 is shown in Table 1.

Operation 3: Correction and Final Result

We can calculate Q and R from the following equations,

$$Q_b = e, \qquad Q = Q_a - e, \tag{16}$$

$$R = g + R_{sh} \tag{17}$$

A correction step is needed when the value of $g + R_{sh}$ is greater than the divisor, B. Then

$$Q_b = e - 1, \qquad Q = Q_a - e + 1, \tag{18}$$

$$R = g + R_{sh} - B \tag{19}$$

Table 1. *The LUT for divisor = 13.*

Inputs		Outputs	
B	R_c	e	g
13	12	5	5
13	11	5	10
13	10	4	2
13	9	4	7
13	8	4	12
13	7	3	4
13	6	3	9
13	5	2	1
13	4	2	6
13	3	2	11
13	2	1	3
13	1	1	8
13	0	0	0

3.5. An Illustrative Example of the Algorithm

$$\frac{A}{B} = \frac{65500}{192}$$

As $128 < B < 256$, key = 128. For every nested division, the values that are assigned as unknowns will be post-calculated during the backward steps.

block1

$$\begin{aligned} A &= 65500 & Q &= unknown \\ B &= 192 & R &= unknown \\ Q1_{sh} &= 511 & Q1_c &= unknown \\ R1_{sh} &= 92 & R1_c &= unknown \end{aligned}$$
$$Q1_{sh} > B$$

block2

$$\begin{aligned} Q1_{sh} &= 511 & Q1_c &= unknown \\ B &= 192 & R1_c &= unknown \\ Q2_{sh} &= 3 & Q2_c &= 0 \\ R2_{sh} &= 127 & R2_c &= 3 \\ Q2_{sh} < B \quad &\rightarrow & \therefore Q2_c &= 0, R2_c = Q2_{sh} = 3 \end{aligned}$$

Result computation2

Inputs:

$Q2_{sh} = 3, \quad R2_{sh} = 127, \quad Q2_c = 0, \quad R2_c = 3$

$$Step1 : Q_a = Q2_{sh} - Q2_c * x = 3$$
$$Step2 : LUT : \quad \because B = 192, R2_c = 3 \therefore e, g = 1, 0$$
$$Step3 : Q_b = 1, R = g + R2_{sh} = 127 < B$$
$$\therefore Q1_c = Q_a - Q_b = 2, \quad R1_c = 127$$

block2

$$\begin{aligned} Q1_{sh} &= 511 & Q1_c &= 2 \\ B &= 192 & R1_c &= 127 \\ Q2_{sh} &= 3 & Q2_c &= 0 \\ R2_{sh} &= 127 & R2_c &= 3 \\ Q2_{sh} < B \quad &\rightarrow & \therefore Q2_c &= 0, R2_c = Q2_{sh} \end{aligned}$$

block1

$$\begin{aligned} A &= 65500 & Q &= unknown \\ B &= 192 & R &= unknown \\ Q1_{sh} &= 511 & Q1_c &= 2 \\ R1_{sh} &= 92 & R1_c &= 127 \end{aligned}$$
$$Q1_{sh} > B$$

Result computation1

Inputs:

$Q1_{sh} = 511, \quad R1_{sh} = 92, \quad Q1_c = 2, \quad R1_c = 127$

$$Step1 : Q_a = Q1_{sh} - Q1_c * x = 383$$
$$Step2 : LUT : \quad \because B = 192, R1_c = 127 \therefore e, g = 43, 128$$
$$Step3 : Q_b = 43, R = g + R1_{sh} = 220 > B$$
$$Correction : Q_b = 42, R = g + R1_{sh} - B = 28$$
$$\therefore Q = Q_a - Q_b = 341, \quad R = 28$$

block1

$$\begin{aligned} A &= 65500 & Q &= 341 \\ B &= 192 & R &= 28 \end{aligned}$$

4. Results

4.1. Algorithm Performance

The suggested algorithm is designed for integer dividends and divisors. Its performance depends on two main factors, the number of nested divisions, and the size of the LUT. These two factors affect the design's area, delay, and power consumption. The two factors change according to the value of the divisor. Nevertheless, the size of the LUT is proportional to the range and the number of bits of the divisor. Table 2, Table 3, Table 4, and Table 5 show the range of divisors for each number of nested $blocks$ for 8-bit, 16-bit, 32-bit, and 64-bit dividends, respectively. The equation that controls the number of nested $block$s is

$$N = \frac{m}{n-1} - 1 \tag{20}$$

Table 2. *Number of nested $blocks$, N, and its feasible divisor values for 8-bit dividend. m=8, n_{max} = 8, and divisor maximum value = $2^8 - 1$ = 255.*

N	n	Divisor values
1	≥ 5	≥ 16
2	≥ 4	≥ 8
3	≥ 3	≥ 4
7	≥ 2	≥ 2

Table 3. *Number of nested $blocks$, N, and its feasible divisor values for 16-bit dividend. m=16, n_{max} = 16, and divisor maximum value = $2^{16} - 1$ = 65535.*

N	n	Divisor values
1	≥ 9	≥ 256
2	≥ 7	≥ 64
3	≥ 5	≥ 16
5	≥ 4	≥ 8
7	≥ 3	≥ 4
15	≥ 2	≥ 2

Table 4. *Number of nested $blocks$, N, and its feasible divisor values for 32-bit dividend. m=32, n_{max} = 32, and divisor maximum value = $2^{32} - 1$ = 4294967295.*

N	n	Divisor values
1	≥ 17	$\geq$ 65,536
2	≥ 12	$\geq$ 2,048
3	≥ 9	≥ 256
4	≥ 8	≥ 128
5	≥ 7	≥ 64
6	≥ 6	≥ 32
7	≥ 5	≥ 16
10	≥ 4	≥ 8
15	≥ 3	≥ 4
31	≥ 2	≥ 2

Table 5. *Number of nested $blocks$, N, and its feasible divisor values for 64-bit dividend. m=64, n_{max} = 64, and divisor maximum value = $2^{64} - 1$.*

N	n	Divisor values
1	≥ 33	$\geq$ 4,294,967,296
2	≥ 23	$\geq$ 4,194,304
3	≥ 17	$\geq$ 65,536
4	≥ 14	$\geq$ 8,192
5	≥ 12	$\geq$ 2,048
6	≥ 11	$\geq$ 1,024
7	≥ 9	≥ 256
9	≥ 8	≥ 128
10	≥ 7	≥ 64
12	≥ 6	≥ 32
15	≥ 5	≥ 16
21	≥ 4	≥ 8
31	≥ 3	≥ 4
63	≥ 2	≥ 2

As proved in Section 3.4, the size of the LUT depends only on the divisor B and is independent on the dividend. The address of the LUT relies on B, and R_c, while the data contained in this LUT is the values of e, and g as shown in Table 1. Where the number of bits of e, and g are $(n-1)$-bit, and n-bit, respectively. For constant divisor B, the address of the LUT is the value of R_c, so the number of rows of the LUT equals to B, and the row size is $2n-1$. On the other hand, in the case of a variable divisor, for the range of divisors with n bits, the number of rows of the LUT differs according to n, but the size of the data inside it remains $2n-1$. Table 6 presents the number of LUT rows according to the size of the divisor. For more explanation, each divisor value takes 2^n rows in the LUT, so the number of LUT rows, LUT_{rows}, is the summation of all numbers from 1 to 2^n excluding the key numbers.

Table 6. *LUT size for variable divisor, n_{min}=1, and row size is $(2n-1)$ bits.*

n	Number of LUT rows	Number of address bits
≤ 3	21	5
≤ 4	105	7
≤ 5	465	9
≤ 6	2k	11
≤ 7	8k	13
≤ 8	32k	15
≤ 9	128k	17
≤ 10	512k	19
≤ 11	2M	21
≤ 12	8M	23
≤ 13	32M	25
≤ 14	128M	27
≤ 15	512M	29
≤ 16	2G	31
≤ 17	8G	33
≤ 18	32G	35
≤ 19	128G	37
≤ 20	512G	39

The following equation represents the number of rows of the LUT for the division by n-bit divisor:

$$\text{For Variable Divisor} \qquad LUT_{rows} = \sum_{i=1}^{2^n-1} i - \sum_{j=0}^{n-1} 2^j \quad (21)$$

As shown from the table, the size of the LUT is huge for large divisors. So, this algorithm is more efficient for constant division for large numbers. For example, the division by a 14-bit constant value needs a LUT size of $16k \times 27 = 2^n \times (2n-1)$. However, the general division by any 14-bit value needs around 3G LUT size, $128M \times 27$. Also, as B is constant, the address is R_c. Table 7 illustrates the maximum number of rows of the LUT for constant division by an n-bit divisor. The number of LUT's rows for constant divisor B follows the following equation:

$$\text{For Constant Divisor} \qquad LUT_{rows} = B \quad (22)$$

Table 7. *LUT size for constant divisor where the data size is $(2n-1)$ bits.*

n	B	number of LUT rows	address bits
3	7	7	3
4	15	15	4
5	31	31	5
6	63	63	6
7	127	127	7
8	255	255	8
9	511	511	9
10	1023	1k	10
11	2047	2k	11
12	4095	4k	12
13	8,191	8k	13
14	16,383	16k	14
15	32,767	32k	15
16	65,535	64k	16
18	262,143	256k	18
20	1,048,575	1M	20
24	16,777,215	16M	24
28	268,435,455	256M	28
32	4,294,967,295	4G	32

4.2. Hardware Implementation Results

The suggested algorithm is implemented in Zynq UltraScale xczu7ev-ffvc1156-2-e FPGA and simulated in Xilinx Vivado ML version 2022.2 to evaluate its performance. Table 8 demonstrates the utilized resources for the implemented variable and constant division on FPGA. It shows the number of configurable logic blocks (CLB) LUT as Logic, CLB Registers as Flip Flops, Block RAM (RAMB36E2), and DSPs (DSP48E2). After synthesis, the latency of one operation, T, is calculated by multiplying the minimum clock cycle by the number of cycles per division operation. The number of clock cycles per operation equals $N + 1$. It can be noted that the routing adds a considerable part of the total latency. Also, BRAM expresses Block RAM, and N is the number of iterations. The results of dividing a 16-bit dividend by both 7-bit and 8-bit divisors need a ROM size of 32K x 15-bit. Furthermore, the constant numbers are chosen randomly in Table 8 as the algorithm works with any value within bits equal to n with LUT's rows equal to B as illustrated in Table 7.

Table 8. *Performance of the suggested design on Zynq FPGA.*

Divisor		Variable division	Constant Division		
		64 < B < 255	225	1279	4095
m		16	16	16	32
n		$7 \leq n \leq 8$	8	11	12
N		2	2	1	2
FPGA Utilization					
LUT		335	147	68	152
BRAM		30	0.5	1.5	3
Register		52	40	43	76
DSPs		1	1	1	2
Carry8		11	9	9	19
Latency	Logic:	3.19	3.49	1.66	4.66
(ns)	Route:	4.69	2.23	1.04	1.68
	Total:	7.88	5.72	2.7	6.34
Freq. (MHz)		126	174	370	157
T (ns)		23.64	17.16	5.4	19.02

5. Comparison and Discussion

This section discusses the implementation of the state-of-the-art. Table 9 compares the different procedures according to method complexity and rigidity. The software/hardware pre-processing, algorithm mathematical operations (adders and multipliers), and LUTs measure the complexity. The maximum number of bits for the dividend, divisor, and accuracy express the rigidity. In the Table, the (Sug.) abbreviates the suggested algorithm, the (Pre-) is the abbreviation of pre-processing, the (Acc./App.) is the abbreviation of accurate/approximate algorithm, and the LUTs size is expressed by the number of row $\times$ number of columns. Also, the values of m and n are the maximum value.

Lemire et al. in 2021 [25] use a complicated pre-processing software algorithm that estimates $(1/B)$. Then they calculate the quotient using the multiply-add operation and the remainder using two successive multiplication operations. They present a software algorithm that accelerates the division operation. Their results are accurate only if the inverse of B can be represented by (c/M) where M is a power-of-2 number, and c equals M/B. Their technique is valid for certain relations between c/M and $1/B$; this relation is called the bound, so the authors optimize and generalize the bounds previously proposed for this technique in the literature. A similar idea is provided by Nigel Jones in his blog entry [29] as he pre-calculated $(1/B)$ using a software calculator. Also, a required parameter labeled as S is computed using successive shift, increment, then compare operations. Even though his

method uses only one 32-bit × 32-bit multiplication operation, for a 16-bit dividend, it is approximate and has a maximum dividend value of 16-bit.

Constant division using the add-shift methodology is hardware implemented in 2015 by Yassin et al. [16]. It is for specific divisors 3, 5, 6, 7, and 9, with 13-bit dividends; for divisor 3, the dividend is up to 16. They use five successive 16-bit additions to generate the quotient and remainder. Their simulation results take seven clock cycles for the division operation. Furthermore, Ugurdag et al. [17], in 2017, implemented the division by small constant integer. They depend on LUT, and have $(m - n)$ LUT access. its LUT size is $(m + n) \times (m + n)$.

Table 9. *Comparison between different division procedures (A/B).*

	m ‖ n	Pre-	Acc. ‖ App.	LUT size	Mathematical operations
[25]	N/A	$c/M = 1/B$	App.	-	S/W algorithm
[29]*	16‖16	(1/n)	App.	n × 32 & n × 16	32-bit multiplier
[17]	128‖6	-	Acc.	$(m + n) \times (m + n)$	-
[16]	13 ‖ 4	-	Acc.	-	5-successive 16-bit adders
Sug.	no ** limits	-	Acc.	2n × (2n-1)	$\approx \frac{m}{n}$ multipliers (m-2n)-bit

*pre-processing and LUTs are alternatives.

**Area, delay, and LUT size are proportional to m/n as stated in Tables 3 to 6.

6. Conclusions

The suggested approach calculates an accurate quotient and remainder while decreasing the complexity of the division operation. This technique depends on recursive division. It can be applied to a wide range of operands. It is convenient for software-based applications, hardware-based applications, and embedded systems. For hardware implementation, its area is relatively large in some cases related to the difference between the number of bits of the dividend and the divisor, and the divisor's size. That is because it uses a LUT size proportional to the size of the divisor for constant division and proportional to the number of variable divisors for variable division. However, The algorithm proves high performance for large divisors when the difference between the dividend and the divisor size is relatively small. Also, as future work, the utilization of the LUT can be replaced by an arithmetic circuit that calculates the variables stored in the LUT. Hence, lower hardware requirements are put forward at the same time.

Abbreviations

ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction Processor
CLB	Configurable Logic Blocks
CPU	Central Processing Unit
DSA	Digital Signature Algorithm
DSP	Digital Signal Processing
FPGA	Field Programmable Gate Arrays
GCD	Greatest Common Divisor
GSA	Gold-Schmidt Algorithm
ICD	Integer Constant Division
LSB	least significant bit
LUT	Look-Up Table
MSB	Most Significant Bit
NRA	Newton-Raphson Algorithm
SEA	Series Expansion Algorithm
RSA	Rivest-Shamir-Adleman cryptosystem
cryptosystem	
SoC	System On Chip
SR	Shift-Right
SRT division	Sweeney, Robertson, and Tocher division
TSEA	Taylor Series Expansion Algorithm
VLSI	Very Large Scale Integration

Conflicts of Interest

The authors declare no conflicts of interest.

References

[1] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "Falcon: Honest-majority maliciously secure framework for private deep learning," Proceedings on Privacy Enhancing Technologies, vol. 2021, pp. 188-208, 2020.

[2] S. Ioffe and C. Szegedy, "Batch normalization: accelerating deep network training by reducing internal covariate shift," in Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37, ICML'15, p. 448-456, JMLR.org, 2015.

[3] S. Josefsson and I. Liusvaara, "Rfc 8032: Edwards-curve digital signature algorithm (eddsa)," Internet Research Task Force (IRTF), 2017.

[4] T. Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," IEEE Transactions on Information Theory, vol. 31, no. 4, pp. 469-472, 1985.

[5] H. T. Sihotang, S. Efendi, E. M. Zamzami, and H. Mawengkang, "Design and implementation of rivest shamir adleman's (rsa) cryptography algorithm in text file data security," in Journal of Physics: Conference Series, vol. 1641, IOP Publishing, 2020.

[6] D. Cavagnino and A. E. Werbrouck, "Efficient algorithms for integer division by constants using multiplication," The Computer Journal, vol. 51, no. 4, pp. 470-480, 2008.

[7] R. Vemula and K. M. Chari, "A review on various divider circuit designs in vlsi," in 2018 Conference on Signal Processing And Communication Engineering Systems (SPACES), pp. 206-209, 2018.

[8] X. Wei, Y. Yang, and J. Chen, "A low-latency divider design for embedded processors," Sensors, vol. 22, no. 7, p. 2471, 2022.

[9] P. K. Meher and T. Stouraitis, Arithmetic Circuits for DSP Applications. Wiley-IEEE Press, 2017.

[10] S. Deshpande, S. M. d. Pozo, V. Mateu, M. Manzano, N. Aaraj, and J. Szefer, "Modular inverse for integers using fast constant time gcd algorithm and its applications," in 2021 31st International Conference on Field-Programmable Logic and Applications (FPL), pp. 122-129, 2021.

[11] C.-Y. Tsai, M.-H. Fan, and C.-H. Huang, "Vlsi circuit design of digital signal processing algorithms using tensor product formulation," Computer Science and Technology, vol. 21, 2008.

[12] P. Montuschi, J. Bruguera, L. Ciminiera, and J.-A. Pieiro, "A digit-by-digit algorithm for mth root extraction," Computers, IEEE Transactions on, vol. 56, pp. 1696-1706, 01 2008.

[13] X. Yang and M. Sun, "Theoretical convergence analysis of a general division-deletion algorithm for solving global search problems," Journal of Global Optimization, vol. 37, p. 27, Jan 2007 2007/01//. Copyright - Springer Science+Business Media B. V. 2007.

[14] N. Aggarwal, K. Asooja, S. S. Verma, and S. Negi, "An improvement in the restoring division algorithm (needy restoring division algorithm)," in 2009 2nd IEEE International Conference on Computer Science and Information Technology, pp. 246-249, 2009.

[15] L. Chen, J. Han, W. Liu, and F. Lombardi, "Design of approximate unsigned integer non-restoring divider for inexact computing," in Proceedings of the 25th Edition on Great Lakes Symposium on VLSI, GLSVLSI ¡¯15, (New York, NY, USA), p. 51-56, Association for Computing Machinery, 2015.

[16] F. M. Yassin, N. M. Syamimi, A. M. bin Manah, and Z. A. Omar, "Rounded constant division via add-shift in verilog," International Journal of Science, Environment and Technology, 2015.

[17] H. F. Ugurdag, F. de Dinechin, Y. S. Gener, S. Gören, and L.-S. Didier, "Hardware division by small integer constants,' IEEE Transactions on Computers, vol. 66, no. 12, pp. 2097-2110, 2017.

[18] S. Obermann and M. Flynn, "Division algorithms and implementations," IEEE Transactions on Computers, vol. 46, no. 8, pp. 833-854, 1997.

[19] S. Dixit and M. Nadeem, "Fpga accomplishment of a 16-bit divider," Imperial journal of interdisciplinary research, vol. 3, 2017.

[20] K. J. N. S. Bhargav, S. Palisetti, and M. Rao, "A newton raphson method based approximate divider design for color quantization application," in 2021 18th International SoC Design Conference (ISOCC), pp. 115-116, 2021.

[21] M. P. Vestias and H. C. Neto, "Revisiting the newtonraphson iterative method for decimal division," in 2011 21st International Conference on Field Programmable Logic and Applications, pp. 138-143, 2011.

[22] J. Wei, A. Kuwana, H. Kobayashi, and K. Kubo, "Revisit to floating-point division algorithm based on taylor-series expansion," in 2020 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), pp. 240-243, 2020.

[23] U. S. Patankar and A. Koel, "Review of basic classes of dividers based on division algorithm," IEEE Access, vol. 9, pp. 23035-23069, 2021.

[24] N. Arya, T. Soni, M. Pattanaik, and G. Sharma, "Energy efficient logarithmic-based approximate divider for asic and fpga-based implementations," Microprocess. Microsyst., vol. 90, apr 2022.

[25] D. Lemire, C. Bartlett, and O. Kaser, "Integer division by constants: optimal bounds," Heliyon, vol. 7, p. e07442, June 2021.

[26] G. Schewior, H. Flatt, C. Dolar, C. Banz, and H. Blume, "A hardware accelerated configurable asip architecture for embedded real-time video-based driver assistance applications," in 2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, pp. 209-216, 2011.

[27] H. F. Ugurdag, A. Bayram, V. E. Levent, and S. Gören, "Efficient combinational circuits for division by small integer constants," in 2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH), pp. 1-7, 2016.

[28] M. McNerney and E. Papadopoulos, "Hacker's delight: Law firm risk and liability in the cyber age," American University Law Review, vol. 62, pp. 1243-1269, 2013. Copyright - Copyright American University Law Review 2013.

[29] N. Jones, "Division of integers by constants." https://embeddedgurus.com/stack-overflow/2009/06/-url-division-of-integers-by-constants/, 2009. Accessed: 2023-01-1.

[30] S. Deshpande, S. M. d. Pozo, V. Mateu, M. Manzano, N. Aaraj, and J. Szefer, "Modular inverse for integers using fast constant time gcd algorithm and its applications," in 2021 31st International Conference on Field-Programmable Logic and Applications (FPL), pp. 122-129, 2021.

[31] D. J. Bernstein and B.-Y. Yang, "Fast constant-time gcd computation and modular inversion," IACR Transactions on Cryptographic Hardware and Embedded Systems, vol. 2019, p. 340-398, May 2019.

[32] A. Greuet, S. Montoya, and C. Vermeersch, "Quotient approximation modular reduction," 2022 IEEE 29th Symposium on Computer Arithmetic (ARITH), pp. 103-110, 2022.