A NOVEL QUOTIENT PREDICTION FOR FLOATING-POINT DIVISION

PHAM TRAN BICH THUAN

Office of Academic Affairs, Industrial University of HoChiMinh City,
phamtranbichthuan@iuh.edu.vn

Abstract. At present, floating-point operations are used as add-on functions in critical embedded systems, such as physics, aerospace system, nuclear simulation, image and digital signal processing, automatic control system and optimal control and financial, etc. However, floating-point division is slower than floating-point multiplication. To solve this problem, many existing works try to reduce the required number
of iterations, which exploit large Look Up Table (LUT) resource to achieve approximate mantissa of a quotient. In this paper, we propose a novel prediction algorithm to achieve an optimal quotient by predicting certain bits in a dividend and a divisor, which reduces the required LUT resource. Therefore, the final quotient is achieved by accumulating all predicted quotients using our proposed prediction algorithm. The experimental results show that only 3 to 5 iterations are required to obtain the final quotient in a floating-point division computation. In addition, our proposed design takes up 0.84% to 3.28% (1732 LUTs to 6798 LUTs) and 5.04% to 10.08% (1916 (ALUT) to 3832 (ALUT)) when ported to Xilinx Virtex-5 and Altera Stratix-III FPGAs, respectively. Furthermore, our proposed design allows users to track remainders and to set customized thresholds of these remainders to be compatible with a specific application.

Keywords. Floating-point number, Floating-point Division, FPU, FPGA, LUT, embedded system.

1. INTRODUCTION

Floating-point numbers can assist to obtain a dynamic range of representable real numbers without scaling operands [1][2][3]. In order to accelerate operations using floating-point numbers, Floating-Point Unit (FPU) is implemented and embedded into the IBM System/360 Model 91, a supercomputer in the mid-1960s, which consists of two floating-point units [3]. FPUs are more expensive and slower than Central Processing Units (CPUs). To reduce these drawbacks, some researches have been carried on to accelerate the FPU through speeding up floating-point computations, such as addition, subtraction, multiplication and division on Field-Programmable-Gate Arrays (FPGA) [4][5] or on Application-Specific Integrated Circuit (ASIC) [6][7].

An ASIC is an integrated circuit (IC) customized for a particular application rather than a general-purpose application. However, a design using ASIC is costly and inflexible to be updated. Compared with this, FPGA is a suitable platform due to its capacities of being easily reconfigured and being upgraded without further cost. Implementation of complex floating-point applications in a single FPGA is possible due to the high integration density of current nanometer technologies. FPGA based floating-point computations have been proposed in [4] and [5].

Compared with basic floating-point operations, such as addition, subtraction and multiplication, floating-point division is the most complex operation among them. In a floating-point division, mantissas or significands of two operands are divided and exponents of these two operands are subtracted. In some cases, a remainder is needed according to the requirement of applications or users who might want to monitor results of the computation. In [1],[2] and [3], the production of the remainder is handled by the software. 'DIV' and 'MOD' commands are used to execute the division and to generate the quotient and the remainder, respectively.

The straightforward method to speed up floating-point division is the digit-recurrent division algorithm, which calculates the quotient using an iterative architecture and generates each quotient per iteration. A quotientdigit selection function is used in each iteration to determine the quotient. In this algorithm, the total iterative number is n if the quotient is n-bits. Another method to speed up floating-point division is the high-radix Sweeney, Robertson and Tocher (SRT) algorithm [1][2][3]. In this

algorithm, each quotient digit is represented by a signed digit $\{\bar{\alpha}, \overline{\alpha-1}, \cdots, \bar{1}, 0, 1, \cdots, \alpha\}$, where $\left\lceil \frac{1}{2}(\beta-1) \right\rceil \leq \alpha \leq (\beta-1)$ and β is the radix value ($\beta = 2^m$). The total iterative number is n/m. The disadvantages of this SRT method are that the divisor must be normalized (MSB equals to 1) before the division, and the final quotient is represented by sign-digit number (SD). Since each digit represented by the SD number requires a signed bit to indicate whether it is positive or negative, this leads to using extra bits. Therefore, there needs an extra function to convert the number represented by SD to the normal binary number.

As discussed above, the SRT division algorithm for floating-point division is well investigated [8]. However, the disadvantage of this algorithm is large latency and it only can achieve less than 10 bits per cycle [9]. Another research extends a dedicated floating-point multiplier to support the division. The disadvantages of this extension are that it lacks of the remainder and the rounding process is complicated [9]. To solve these issues, the designer should rewrite the programming code [7]. Pineiro and Bruguera propose LUT approximations and Taylor-series approximations schemes to reduce the number of iterations by the use of approximate quotient method [10]. But, their method only focuses on software platform. Therefore, the procedure of the computation is complicated [11]. Amin and Shinwari propose to exploit variable latency dividers to generate the appropriate number of quotient bits based on different exponents [12]. On the other hand, Kwon and Draper proposed a fused floating-point multiplication/division/squaring based on the Taylor-series algorithm [13]. However, the speed of the proposed method could not meet the requirements for mobile applications [14]. The high-radix algorithm is proposed to reduce the computational time [15][16][17][18]. The disadvantages of this method are: (1) the required number of iteration is large; (2) the remainder should be normalized when its Most Significance Bit (MSB) equals to 1; (3) an additional computation is required to determine the number of the quotient's bits in each iteration.

The number of iterations in these methods above is fixed, which depends on the length of significands. Different to these methods, some methods employ an optimal function to obtain the final result. They are Co-Ordinate Rotation-Digital-Computer (CORDIC), Newton-Raphson-Base division, Genetic -Algorithm (GA), and Chemical-Reaction-Optimization (CRO). CORDIC method uses only shifting, addition and LUT modules to transform an expected angle of hyperbolic and trigonometric functions to a corresponding set of binary numbers. The Newton-Raphson-Base division is a technique, which uses iterative architecture to obtain roots [2][19]. The CRO is proposed based on the GA method [20]. The GA and the CRO methods only can handle randomly selected values, in which the computation must be repeated until a best adjacent result is achieved. They also exploit iterations to obtain the best adjacent value based on a data set. Therefore, larger memory resource and higher speed are required for a system.

In this paper, we propose to enhance the convergence method to achieve the final result based on CORDIC. We also improve the Newton-Raphson method to achieve the best adjacent result based on the GA and the CRO methods. If the best adjacent result is achieved, the computation of the proposed method will cease, which does not depend on the length of significands of a dividend and a divisor. The final quotient is achieved by accumulating all the predicted quotients in each iteration. Furthermore, the proposed algorithm allows users to track the remainder during the computation. This is to say, the remainder can be set to the customized threshold values by users. Our proposed algorithm improves the scalability of predicted values stored in LUT (using 256 to 4096 elements in LUT) and the scalability of adjusted exponent values (using NOT gate & AND gate), which is based on our previous work [21]. Therefore, the proposed design achieves relatively accurate predicted quotient in each iteration. The experimental results show that the proposed computation of the quotient is faster than the existing methods using LUT.

The rest of this paper is organized as follows: Section 2 presents floating-point numbers and digit recurrence division algorithm. Section 3 illustrates the proposed algorithm. Section 4 shows experimental results. Section 5 draws the conclusion.

2. PRELIMINARIES

A floating-point number can be represented in various formats. Also, the results of floating-point computations are imprecise. This is to say, each floating-point related computations is approximate. Transformation among different formats of the input data will be time-consuming. Therefore, the Institute of Electrical and Electronics Engineers (IEEE) introduced the IEEE 754 standard in 1985, the IEEE 854 standard in 1987, and the IEEE 754 standard in 2008 [2]. Rounding methods are also presented in [1][2][3][14] to solve the approximation of floating-point computations.

We will present floating-point division algorithm in the followings. A typical $n-bits$ floating-point number consists of $1-bit$ sign (S), $e-bits$ exponent (E) and $k-bits$ unsigned fraction (M). The length of this number is $n=k+e+1$.

A floating-point number can be represented by Equation (1):

$$F=(-1)^S\cdot M\cdot \beta^E \tag{1}$$

Where $(-1)^0=1$ and $(-1)^1=-1$. β is the base of the exponent E. $M=\sum_{i=1}^{k} b_i r^{-1}$ and $r^{-k}\leq M\leq 1-r^{-k}$.

Similarly, floating-point numbers F_1 and F_2 can be represented as:

$$F_1=(-1)^{S_1}\cdot M_1\cdot \beta^{E_1-bias} \tag{2}$$

$$F_2=(-1)^{S_2}\cdot M_2\cdot \beta^{E_2-bias} \tag{3}$$

Where, *bias* is a constant number.

Suppose that the result of F_1 divided by F_2 is:

$$F_3=(-1)^{S_3}\cdot M_3\cdot \beta^{E_3-bias} \tag{4}$$

Where $M_3=\frac{M_1}{M_2}$ and $E_3=E_1-E_2$.

Given a dividend (M_1), a divisor (M_2), a quotient and remainder (M_R) should satisfy Equation (5) [1][2][3]:

$$M_1=M_3.M_2+M_R \qquad (M_1<M_2) \tag{5}$$

At the i^{th} iteration, a remainder is computed as shown in Equation (6):

$$m_{3.i}=\begin{cases}1, & if\ 2.r_{i-1}\geq M_2\\ 0, & if\ 2.r_{i-1}<M_2\end{cases} \qquad (i\in 0,1,2,\cdots,k) \tag{6}$$

Where $M_3=0.m_{3.1}m_{3.2}\cdots m_{3.k}$, $\beta=2$, k is the length of the unsigned fraction *(M)*. The computation of the remainder at i^{th} iteration is as follows:

$$r_i=2.r_{i-1}-m_{3.1}\cdot M_2 \qquad (i\in 1,2,\cdots,k) \tag{7}$$

Where r_i is the remainder at the i^{th} iteration and r_{i-1} is the remainder at the $(i-1)^{th}$ iteration. The remainder at the first iteration is $r_0=M_1$. The final remainder can be represented as $M_R=r_k.2^{-k}$.

The total number of iteration depends on the formats of the floating-point number. These formats are single precision, double precision and double extended.

The architecture of floating-point division is shown in Figure 1. First, two floating-point operands are unpacked, which will separate the sign, the exponent, and the significand for each operand. It also converts these operands to the internal format. The intermediate significand and the intermediate exponent are computed through several steps: dividing significands, normalizing significands, rounding significands, subtracting exponents, and adjusting exponents. The final result is packed into the appropriate format, which combines the sign, the exponent and the significand together. The sign of the quotient is calculated by XORing these operands' signs.

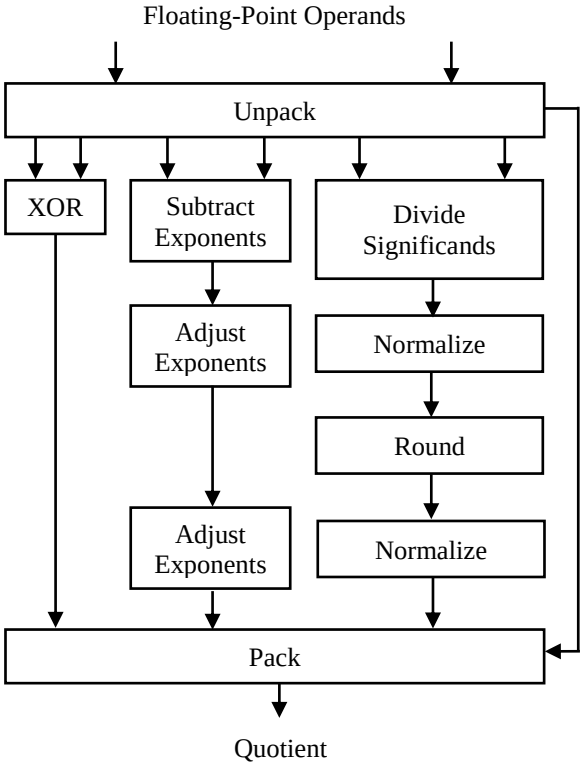

Figure 1: Block diagram of the Floating-point division algorithm

3. THE PROPOSAL ALGORITHMS TO ACCELERATE FLOATING-POINT DIVISION

3.1 The proposed Quotient Prediction Algorithm

Given a dividend F_1 and a divisor F_2, Equation (8) shows how to obtain the quotient F_3 and the remainder F_R .

$$F_1 = F_3 \cdot F_2 + F_R \tag{8}$$

Where F_1, F_2, F_3 and F_R are floating-point numbers. They are defined as $F_1 = (-1)^{S_1} \cdot M_1 \cdot \beta^{E_1}$, $F_2 = (-1)^{S_2} \cdot M_2 \cdot \beta^{E_2}$, $F_3 = (-1)^{S_3} \cdot M_3 \cdot \beta^{E_3}$ and $F_R = (-1)^{S_R} \cdot M_R \cdot \beta^{E_R}$, where S_1, S_2, S_3 and S_R are sign bits. M_1, M_2, M_3 and M_R are mantissas, and E_1, E_2, E_3 and E_R are exponents.

Equation (8) can be rewritten as:

$$F_1 = n.F_2 + F_1' \tag{9}$$

Where n is a fixed coefficient, and it is represented as $n = (-1)^{S_n} \cdot M_n \cdot \beta^{E_n}$ (S_n is a sign bit, M_n is a mantissa and E_n is an exponent). There should exist F_1', which is represented as a complement number of F_1. $F_1' = (-1)^{S_1'} \cdot M_1' \cdot \beta^{E_1'}$, where S_1' is a sign bit, M_1' is a mantissa and E_1' is an exponent.

If left and right sides of Equation (9) are divided by F_2, we can obtain:

$$\frac{F_1}{F_2} = \frac{n.F_2 + F_1'}{F_2} = n + \frac{F_1'}{F_2} \tag{10}$$

Equation (10) can be rewritten as :

$$\frac{F_1}{F_2} = \frac{n_1.F_2 + F_{1,1}'}{F_2} = n_1 + \frac{F_{1,1}'}{F_2} = n_1 + \frac{n_2.F_2 + F_{1,2}'}{F_2} = n_1 + n_2 + \frac{F_{1,2}'}{F_2} = \cdots = n_1 + n_2 + n_3 + \cdots + n_l + \frac{F_{1,l}'}{F_2} \tag{11}$$

Where $n_i (i \in 0, 1, \cdots, l)$ is the fixed coefficient at the i^{th} iteration. $F_{1,\mathrm{i}}'(i \in 0, 1, \cdots, l)$ is the corresponding complement number of $F_{1,i}$ at the i^{th} iteration. l is the total number of iterations.

From Equations (9) and (11), the final quotient and the final remainder can be computed as follows:

$$F_3 = \sum_{i=1}^{l} n_i \; ; \qquad F_R = F'_{1,l} \tag{12}$$

l is independent of single precision, double precision and double extended formats, but it depends on the expected remainder set by users. The computational time of the division varies due to different coefficients n (n is a prediction) set by users. Unlike the traditional floating-point division computation, the final quotient of our proposed algorithm is the subtotal of partial predicted quotients at each iteration. The number of iterations is determined by the accuracy of the prediction and the expected remainder set by users.

Algorithm 1 shows the proposed quotient prediction algorithm.

Algorithm 1: Proposed floating-point division algorithm

Input: Dividend F_1, Divisor F_2

Output: Quotient F_3, Remainder F_R

1. 1^{st} iteration: $F'_{1,i} = F_1$; i^{th} iteration: $F'_{1,i} = F_R$
2. Generate predicted quotient's coefficient pr_i
3. Adjust pr_i to obtain predicted quotient n_i
4. Obtain quotient $F_{3,i}$ (with $F_{3,0} = 0$) and remainder $F'_{1,i+1}$ at the i^{th} iteration

$$F_{3,i} = F_{3,i-1} + n_i; \;\; F'_{1,i+1} = F'_{1,i} - n_i \cdot F_2; \;\; F_r = F'_{1,i+1}$$

5. Compare the new remainder $F'_{1,i+1}$ with the pre-set remainder. If they are the same go to step 1, else go to step 6.
6. Compute

$$F_3 = F_{3,i}; \quad F_r = F'_{1,i+1}$$

This algorithm consists of four functions. They are: A.Predicting the quotient's coefficient function; B.Adjusting the quotient's coefficient to obtain the predicted quotient function; C.Obtaining the quotient value and the remainder value at each iteration; D.Finishing the process and selecting appropriate sign for the final quotient and the final remainder. Function A is used to obtain the quotient's coefficient pr_i in Equation (13). The normalization of Function B is to meet the standard formats of IEEE (single precision, double precision or double extended) and to ensure that the remainder must be positive or equal to zero after the operations in each iteration. Function C helps to obtain the final quotient F3 using Equation (12) and to obtain a new dividend for the next iteration, which is the remainder in this iteration. Function D stops to retrieve quotient and generates results of division.

We will detail these function in the following.

A. Predicting the quotient's coefficient pr_i function:

Predicting the quotient's coefficient (pr_i) function can predict the coefficient pr_i at each iteration, which is stored in an LUT. This LUT is used to store left significant bits of a dividend $F'_{1,i}$ and a divisor F_2 which are represented using IEEE floating-point format [1][2]. In this format, the first bit in the mantissa of $F'_{1,i}$ and F_2 equals to 1. Thus, it is unnecessary to consider the first bit of $F'_{1,i}$ and F_2. We combine left significant m-bits of $F'_{1,i}$ with left significant m-bits of F_2. When m equals to 5, 5-bits of F_1 and 5-bits of F_2 are combined to form one byte (regardless of the first bit ('1') of both), which indicates 256 addresses that can be stored in an LUT with 256 elements. When m equals to 7, 7-bits of F_1 and 7-bits of F_2 are combined to form 14-bit, which indicates that 4096 addresses can be stored in an LUT with 4096 elements.

One element, b, in an LUT is 8-bits width, which is defined as $b = \{b_7, b_6, \cdots, b_0\}$. Among these, b_0 is an extended exponent and the rest 7-bit are the mantissa of this quotient. During a division operation, it automatically uses the first m-bits of $F'_{1,i}$, m-bits of F_2 to generate the address of these elements (m is 5-bit or 7-bit). INT operation is to obtain the integer part of the floating point digital number. MOD is to obtain the decimal fraction part of the floating point digital number.

The predicted quotient's coefficients pr_i is retrieved by the following equations:

$$x_0 = \frac{\sum_{i=0}^{m} F_{1.i} \times 2^{-(i+1)}}{\sum_{i=0}^{m} F_{2.i} \times 2^{-(i+1)}} \times 2 \quad ; \qquad (m = 5/7) \tag{13a}$$

$$x_i = MOD(x_{i-1}) \times 2 \qquad ; \quad (i \in 1,2,\cdots,7) \tag{13b}$$

$$b_i = INT(x_i) \qquad ; \quad (i \in 1,2,\cdots,7) \tag{13c}$$

Where INT operation is to obtain the integer part of the floating point digital number. And MOD operation is to obtain the decimal fraction part of the floating point digital number.

Algorithm 2 shows predicting the quotient's coefficients algorithm.

Algorithm 2: Predicting the quotient's coefficient pr_i algorithm

Input: m-bits of $F'_{1,i}$, m-bits of F_2

Output: Predicted quotient's coefficient pr_i

1. Combine m-bits of $F'_{1,i}$, m-bits of F_2 to form an element's address
2. Obtain an element from LUT, which has the corresponding element's address
3. Assign this element's value to pr_i

Algorithm 2 shows that there are three steps to predict quotient's coefficient. The purpose of step 1 is to obtain an address. According to this address, the algorithm will obtain a corresponding element's address in an LUT. Then, the outcome of pr_i is achieved in step 3.

B. Adjusting predicted quotient (n_i) function:

Adjusting predicted quotient (n_i) function consists of two sub-functions: Adjusting mantissa's prm_i function and adjusting exponent's pre_i function. 'Adjusting quotient' is to adjust the values of the quotient's coefficient pr_i (including the mantissa prm_i and the exponent pre_i) to obtain the predicted quotient (n_i). In the adjusting mantissa's prm_i function, in order to smooth computation, the mantissa's prm_i must be post-normalized. This normalization is to add one or several 0's to the end of this mantissa, which makes it to be compatible with the standard format of IEEE. For example, if we use single precision format, the length of mantissa is 23-bit. The initial length of the mantissa in the proposed algorithm is 5 (or 7) bits, therefore 18 (or 16) zeros must be added to the end of the mantissa. In adjusting exponent's pre_i function, pre_i is obtained between the mantissas of the dividend $F'_{1,i}$ and the divisor F_2 are not taken into consideration. However, it is not the final predicted value. In order to obtain an accurate final value, the exponent of the predicted quotient needs to be formulated according to Equation (14).

$$Exponent\ pre_i = (expo(F'_{1,i}) - expo(F_2)) \pm b_7 \tag{14}$$

Where $expo(F'_{1,i})$ is the exponent of the dividend $F'_{1,i}$, $expo(F_2)$ is the exponent of the divisor F_2 and b_7 is the 7^{th} bit in the LUT element.

The remainder value must be positive or equals to zero after the operation of each iteration. To ensure this, Equation (14) shows the required operation. $\overline{F'_{1,i.7}}.F_{2.7}$ with 1-bit, is called "adjust" value. When the 7^{th} bit of mantissas, $F'_{1,i}$ and F_2, are equal, $F'_{1,i.7}$ and $F_{2.7}$ should be scale to a correct quotient to ensure that the value of the remainder is positive. If $F'_{1,i.7}$ is larger than or equals to $F_{2.7}$ he "adjust" value equals to 0, else -1.
Equation (15) can be rewritten as:

$$Exponent\ pre_i = \left(expo(F'_{1,i}) - expo(F_2)\right) \pm pr_{i.7} + \left(\overline{F'_{1,i.7}} \cdot F_{2.7}\right) \tag{15}$$

Algorithm 3 shows the adjusting quotient's coefficient pr_i algorithm to obtain the predicted quotient n_i.

Algorithm 3: Adjusting predicted quotient n_i algorithm

Input: Predicted quotient's coefficient n_i, Dividend $F'_{1,i}$, Divisor F_2

Output: Predicted quotient n_i with length's IEEE single/double/extended-precision format

1. Adjust the mantissa prm_i by adding one or several 0's to its end.

2. Adjust exponent pre_i by comparing the $5^{th}/7^{th}$-bit of mantissas $F'_{1,i}$ and F_2. If $F'_{1,i.7}$ by comparing the $F_{2.7}$, the the "adjust" value equals to 0, else -1.

3. Assign adjusted vales to the predicted quotient n_i

In Algorithm 3, there are two main functions. One is used to adjust mantissa value and the other one is used to adjust exponent value of the predicted quotient's coefficient pr_i, which are based on the initial values,
such as the predicted quotient's coefficient pr_i, the dividend $F'_{1,i}$ and the divisor F_2. The mantissa's pr_i will be adjusted in order to be compatible with the length of IEEE standard format. The exponent's pre_i depends $5^{th}/7^{th}$-bit of mantissas, $F'_{1,i}$ and F_2.

C. Obtaining the quotient value and the remainder value at each iteration:

These computations aim to obtain a quotient and a remainder using Equation (12) at the i^{th} iteration. This remainder becomes a dividend at the $(i+1)^{th}$ iteration. Equation (16) is used to obtain the quotient, which is deduced from Equation (13) and (14):

$$F_{3,i} = F_{3,i-1} + n_i \tag{16}$$

Where $F_{3,i}(i \in 0,1,\cdots,l)$ is the quotient of division at the i^{th} iteration ($F_{3,0} = 0$ at the initial iteration). $F_{3,i-1}$ is the quotient at the $(i+1)^{th}$ iteration and n_i is the predicted quotient at the i^{th} iteration.

$$F'_{1,i+1} = F'_{1,i} - n_i.F_2 \tag{17}$$

In Equation (17) , $F'_{1,i+1}$ is a remainder, $F'_{1,i}$ is a dividend, F_2 is a divisor and n_i is the predicted quotient. The process of identifying $F'_{1,i+1}$ occurs at the same time of obtaining $F_{3,i}$.

Algorithm 4 obtains the quotient $F_{3,i}$ and $F'_{1,i+1}$ at the i^{th} iteration.

Algorithm 4: Obtaining quotient $F_{3,i}$ and remainder $F'_{1,i+1}$

Input: Predicted quotient n_i, Dividend $F'_{1,i}$ and Divisor F_2

Output: The quotient $F_{3,i}$ ($F_{3,0} = 0$ at the initial iteration) and remainder $F'_{1,i+1}$

1. Obtain the quotient $F_{3,i}$: $F_{3,i} = F_{3,i-1} + n_i$
2. Obtain the remainder $F'_{1,i+1}$ by equation: $F'_{1,i+1} = F'_{1,i} - n_i.F_2$

D. Ending the process and selecting the appropriate sign for the final quotient and the final remainder:

$F'_{1,i+1}$ is the remainder after the i^{th} iteration and it is compared with the required remainder set by users.

- If the remainder $F'_{1,i+1}$ does not equal to the pre-set remainder, a new iteration will be computed. $F'_{1,i+1}$ will become a new dividend while the divisor will still remain the same as F_2.
- If the remainder $F'_{1,i+1}$ equals to the pre-set remainder, the computation will be terminated. $F'_{1,i+1}$ is the final remainder and $F_{3,i}$ is the final quotient

At the end of this computation, we need to assign a positive or a negative sign for the final quotient and the final remainder.

- If the dividend F_1 and the divisor F_2 are either positive or both negative: the sign of the final quotient is positive.
- If the signs of dividend F_1 and the divisor F_2 are opposite, the sign of the final quotient is negative.
- The sign of the final remainder must be the same sign as the one of the dividend.

Sign bits of the final quotient and the final remainder are computed as follows:

$$sign(quotient) = sign(dividend) XOR\ sign(divisor) \tag{18}$$

$$sign(remainder) = sign(dividend) \tag{19}$$

Figure 2 shows the architecture of the proposed Algorithm 1 using *m*-bit datapath. This architecture ($F_{3,0} = 0$ at the initial iteration) consists of six parts. (1) $F_{1(Original)}$ is the dividend F_1, $F_{2(Original)}$ is the divisor F_2, and $F'_{1,i}$ is the remainder, which becomes a dividend in the next iteration. (2) Multiplexer (MUX2-1) determines to pass through F_1 or $F'_{1,i+1}$. The multiplexer is controlled by signal 'Sel cont'. If 'Sel cont'=0, F_1 is allowed to pass through the multiplexer, else $F'_{1,i+1}$ is allowed to pass through. 'Sel cont' is initialized to 0 at the beginning of this computation. (3) 'Predict quotient's coefficient pr_i' has the same definition as shown in Part A. (4)'Adjust exponent pre_i' and 'Adjust mantissa prm_i' are two functions in 'Adjusting predicted quotient (n_i)' function, which have the same definitions as shown in Part B. (5) Equations (16) and (17) are used to obtain the final quotient and the final remainder at the i^{th} iteration. They have the same definitions as shown in Part C. (6) The result of comparing the final remainder with the pre-set remainder can be used to decide whether to continue or to terminate this computation, which is presented in Part D.

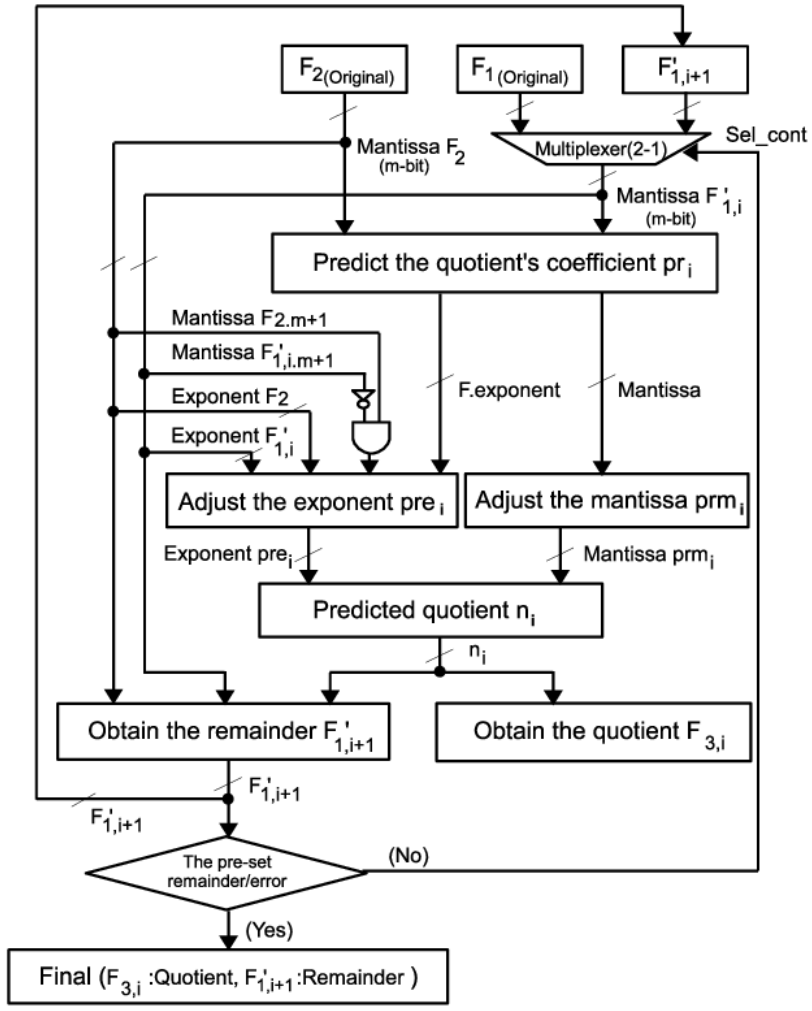

Figure 2: Block diagram of the proposed architecture with m-bit predichtion

3.2 Enhancing the proposed algorithm using FMA instructions

FMA instruction was implemented in 1990 on the IBM RS/6000 processor to facilitate the rounding part of a floating-point division. FMA is suitable for dot products, matrix multiplications, and polynomial computations, etc. Nowadays, FMA is used to accelerate computational speed and to reduce errors for the floating-point division [22][23][24]. Assume that the rounding operations is o, and A, B, C are floating-point numbers. FMA(A, B, C) is represented as o (*A.B + C*). This operation is compatible with the IEEE floating-point format. Therefore, its result must be rounded and normalized [1][2]. Figure 3 shows the

architecture of the extended implementation with FMA for the proposed algorithm. Compared to Figure 2, " Obtain the remainder $F'_{1,i+1}$" is substituted by FMA in Figure 3. In addition, F_2 is substituted by $-F_2$ at the input of FMA. This helps FMA to have a negative input value, which is $FMA(n_i, -F_2, F'_{1,i})$.

Inputs of FMA function are $-F_2$, n_i and $F'_{1,i+1}$. The final result is as follows:

$$F'_{1,i+1} = (-F_2) \cdot n_i + F'_{1,i} = F'_{1,i} - n_i \cdot F_2 \quad (20)$$

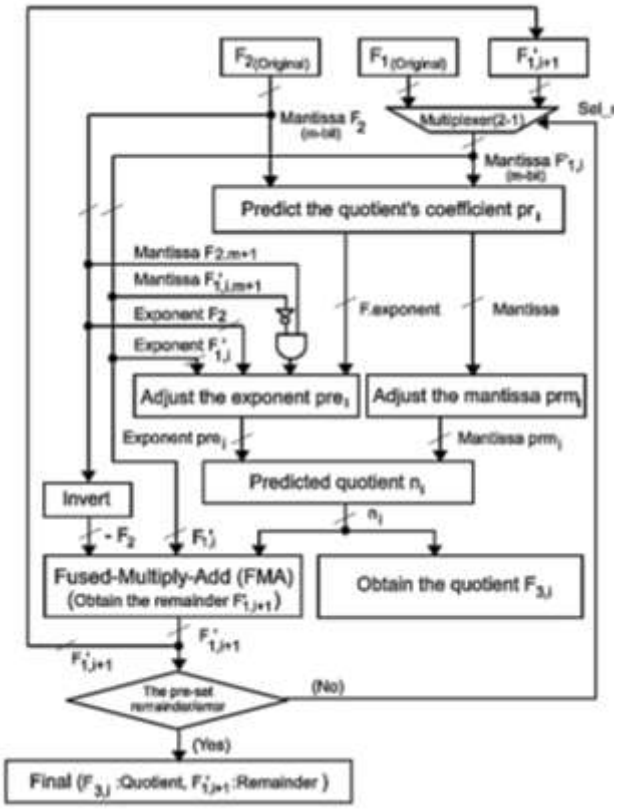

Figure 3: Block diagram of the extended implementation with FMA

4.2 RESULT AND DISCUSSION

The proposed algorithm is implemented on ISE 14.1 of Xilinx Company, Quartus 9.0 of Altera Company and ModelSim 6.5a, which utilizes Verilog, a hardware description language, to describe the algorithm.

Table 1: The results of floating-point division using single precision, double precision and double extended formats on XC5VLX330

Format	**Single Precision**		**Double Precision**		**Double Extended**	
	P5	P7	P5	P7	P5	P7
Frequency (MHz)	193	151	162	131	121	112
Number of Slices	139 (0.07%)	193 (0.09%)	301 (0.15%)	327 (0.16%)	548 (0.26%)	591 (0.28%)
Number of LUTs	1732 (0.84%)	2346 (1.13%)	3728 (1.80%)	4167 (2.01%)	6798 (3.28%)	7687 (3.71%)

P5: 5-bit prediction; P7: 7-bit prediction

The implementation results of the proposed architecture (refer to Figure 2) are presented in Table 1, which include the frequency, the number of slices and LUTs on XC5VLX330 FPGA. These results are

obtained under two cases: (1) left significant 5-bit of the dividend F_1 and the divisor F_2; (2) left significant 7-bit of the dividend F_1 and the divisor F_2;. Table 1 highlights the differences among frequency, the number of slices and LUTs. In addition, three formats i.e. single precision, double precision and double extended precision, are used in these implementations.

From Table 1, when the length of mantissa, exponent and LUT size increases, the occupied area becomes larger and the frequency decreases. However, the increasing degree of area is insignificant, because the proposed design occupies 139 to 591 slices (1732 to 7687 LUTs).

Table 2 shows the frequency, the required number of clock cycles for one iteration as well as the occupied slices and LUTs of our proposed designs on XC5VLX330 FPGA. The results are obtained with pairs of the dividend F_1 and the divisor F_2 using different remainders. For example, mantissa = 1, exponent = -5, -10, -15, - 20 and the pre-set remainder = 0.003125, 0.00097656, 0.000030518, 0.0000009536.

Table 2: Latencies of 5-bit and 7-bit of the dividend and the divisor for prediction on XC5VLX330

Remainder Exponents	**Frequency (MHz)**		**No.Iterations (average)**		**Clock cycles**		**Area Slices**		**Area LUTs**	
	P5	P7	P5	P7	P5	P7	P5	P7	P5	P7
- 5	121	112	4	3	5	5	139	591	1729	7687
- 10	121	110	5	4	5	5	139	591	1729	7688
- 15	120	111	5	4	5	5	141	595	1735	7695
- 20	120	110	5	4	5	5	141	595	1736	7695

P5: 5-bit prediction; P7: 7-bit prediction

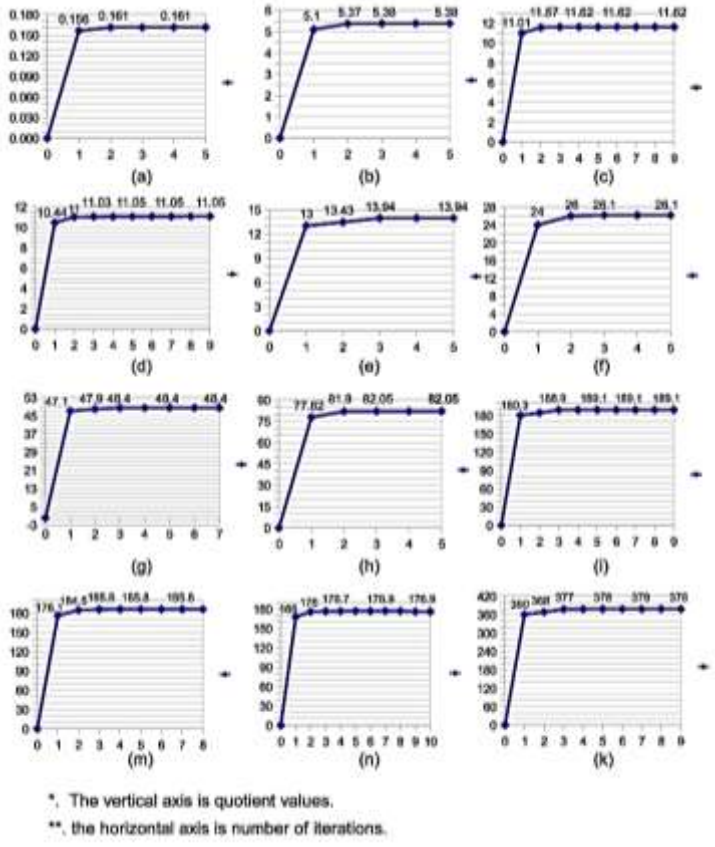

Figure 4: The number of iterations to reach different quotient (pairs of dividend and divisor are randomly selected) (a) Q=0.161247; (b) Q= 5.377; (c) Q= 11.059639; (d) Q= 11.615; (e) Q= 13.94482421875; (f) Q= 26.18; (g) Q= 48.485; (h) Q= 82.05; (i) Q= 176.992; (m) Q= 185.852416; (n) Q= 189.255876608; (k) Q= 378.55.

Figure 4 shows the relationship between the different obtained quotient values and the number of iteration. In Figure 4 from (a) to (l), we randomly choose the pairs of the dividend F_1 and the divisor F_2 based on test vector sets as shown in Table 2. It is obvious that the computed quotient value is close to the required quotient value in the first iteration. It could reach the optimal condition after the second iteration and remains stable for the rest iterations.

From Figure 4 and Table 2, we can draw a conclusion that approximately 3 to 5 iterations on the average are needed to obtain the final quotient. A significant speedup of convergence is achieved in the first two iterations of the computation. The speed of this convergent slows down or remains stable from the third iteration or the fourth iteration onwards. For example, the results of 10 divided by 3 can be 3.3333, 3.333333, or 3.3333333, which depends on the required precision. If the dividend is larger than the divisor, the speed of convergence to obtain the final quotient is faster. In the case that the pre-set reminder is small, the number of iterations to reach the stable state is large, which results in the longer computational time.

Table 3 shows the implementation results of our proposed algorithms on XC5VLX330 and EP3SE50F484C2 FPGAs, respectively. In this particular test, the dividend is $0.100001111.2^7$, the divisor is $0.10010001.2^5$, the required remainder mantissa is 1, and the required remainder's exponent is -5. The results show that the proposed design takes up 0.84% to 3.28% (1732 LUTs to 6798 LUTs) and 5.04% to 10.08% (1916 (ALUT) to 3832 (ALUT)) on XC5VLX330 and EP3SE50F484C2 FPGAs, respectively.

Table 3: The implementation results of the floating-point division on XC5VLX330 and EP3SE50F484C2

	Single precision		Double precision		Double extended	
Platforms	Virtex-5	Stratix III	Virtex-5	Stratix III	Virtex-5	Stratix III
Frequency (MHz)	193	179	162	160	162	158
Number of Slices	139 (0.07%)	1916 (ALUT) (5.04%)	301 (0.15%)	2805 (ALUT) (7.38%)	548 (0.26%)	3832 (ALUT) (10.8%)
Number of LUTs	1732 (0.84%)	1916 (ALUT) (5.04%)	3728 (1.80%)	2805 (ALUT) (7.38%)	6798 (3.28%)	3832 (ALUT) (10.08%)
i (iterations)	5	5	5	5	5	5
Total time (ns)	155	196	182	230	217	263

Table 4 shows a comparison between our proposed algorithm and existing floating-point divisions for double precision. It is quite hard to make a fair comparison due to different algorithms and different platforms used in the existing works. Therefore, we focus on the comparison of number of iterations. The maximum numbers of iterations in [5] and [25] with the non-restoring algorithm and digit-current algorithm are 29 and 55. Compared to them, 13.8% and 54% reduction on maximum number of iterations has been achieved by our algorithm. The maximum number of iterations used in [7] in with Newton Raphson method is 31. Compare with this, our proposed algorithm reduces 19.4% number of iterations. The maximum number of iterations in [8] with SRT method is 40. Our proposed algorithm only requires 25 iterations. In the worst case, the maximum number of iterations of our proposed algorithm is 20% larger than the one in [17] and the same in [6]. This also proves that our proposed algorithm is able to overcome shortcomings of the SRT method, which can efficiently reduce the number of iterations and computational latency. The proposed design takes up 139 to 548 slices on Xilinx Virtex-5 FPGA, which is only 50% to 74% of the designs in [5] and [25] and reduces number of iterations with [26] using CR algorithm.

Table 4: Latency comparisons between this work and previous works

Works	Algorith-ms	Platform	Area	Total of Iteratio-nns	Cycle Time (ns)	Latency (ns)
P.Echeverri'a [5]	NRA	Virtex-4	742 (Slices)**	24 ~ 29	3.6 ~ 2.9	85.7 ~ 114.4
M. Schulte [6]	GA	K7 (FPU)	-	14 ~ 26	-	-
	GA	GS1 (FPU)	-	11 ~ 25	-	-
P. Soderquist [7]	SRT-8/16	-	-	8 ~ 15	7.14	57 ~ 107
	NRM	PA7200(FPU)	-	14 ~ 19	7.14	107
	NRM	PA8000(FPU)	-	31	5	155
S. Oberman [8]	SRT – 1	SPECfp92	-	> 40	-	-
	SRT-2/4/8	SPECfp92	-	4 ~ 40	-	-
T. Lang [17]	SRT - 10	CMOS std (90-nm)	-	20	1	20
M. Baesler [25]	DRA	Virtex-5	55 ~ 261 (Slices)*	1 ~ 55	153.7 ~ 6.8	153.7 ~ 374
Björn Liebig [26]	CR	XC5VFX200T-1	-	10 ~ 57	5	-
This work	PQC	Virtex-5	139 (Slices)*	5 ~ 25	6.2 ~ 5.2	155 ~ 217
			548 (Slices)**			
		Stratix III	1916 (ALUTs)*	5 ~ 25	7.8 ~ 6.2	196 ~ 263
			3832 (ALUTs)**			

-: not supported; *: Single precision; **: Double extended
NRA: Non-restoring Algorithm; GA: Goldschmidt Algorithm
SRT: The high-radix Sweeney, Robertson and Tocher Algorithm
NRM: Newton-Raphson Method; DRA: Digit-Recurrent Algorithm
PQC: Predicting the quotient's coefficient by LUT

5.2 CONCLUSIONS

The floating-point division is the most complicated computations among four floating-point operations, such as addition, subtraction, multiplication and division. In order to reduce the required number of iterations, we focus on the acceleration to obtain the final quotient using the prediction of the quotient at each iteration. In our proposed algorithm, only 3 to 5 iterations are needed in order to reach final quotient in the floating-point division computation. The major advantage of our algorithm is that it is independent on different formats of floating-point number. Moreover, our proposed design utilizes FMA function, which has the advantages of obtaining the remainder easily, avoiding "Normalize" step, and reducing effort in coding. The experimental results show that the proposed design only occupies 0.07% to 0.26% (139 slices to 548 slices) and 5.04% to 10.08% (1916 (ALUT) to 3832 (ALUT)) on Virtex-5 and Stratix-III, respectively. Furthermore, our proposed design reduces the maximum number of iterations to obtain the final quotient, with 26% to 50% reduction of the occupied area compared to the state-of-the-art works.

REFERENCES

[1] I. Koren, Computer Arithmetic Algorithms, AK Peters Ltd, 2002.

[2] J.-M. Muller, N. Brisebarre, F.-D. Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehle, and S. Torres, Handbook of FloatingPoint Arithmetic, Boston-Basel-Berlin, United States, 2009.

[3] T. Lang and M. D.Ercegovac, Digital Arithmetic, Morgan Kaufmann Publishers, 2004.

[4] http://www.xilinx.com/about/all-programmable-leadership/index.htm, 2013.

[5] P. Echeverr´ia and M. L´ opez-Vallejo, Customizing floating-point units for fpgas: Area-performance-standard trade-offs, Microprocessors and Microsystems, Available: www.elsevier.com/locate/micpro, vol. 35, pp. 535–546, 2011.

[6] M. Schulte, D. Tan, and C. Lemonds, Floating-point division algorithms for an x86 microprocessor with a rectangular multiplier, in Computer Design, 2007. ICCD 2007. 25th International Conference on, October 2007, pp.304–310

[7] P. Soderquist and M. Leeser, Division and square root: choosing the right implementation, IEEE Micro, vol. 17, no. 4, pp. 56 –66, July/August 1997.

[8] S. Oberman and M. Flynn, Design issues in division and other floating-point operations, IEEE Transactions on Computers, vol. 46, no. 2, pp. 154 –161, February 1997.

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

[10] J.-A. Pineiro and J. Bruguera, High-speed double-precision computation of reciprocal, division, square root, and inverse square root, IEEE Transactions on Computers, vol. 51, no. 12, pp. 1377 – 1388, December 2002.

[11] D. Wong and M. Flynn, Fast division using accurate quotient approximations to reduce the number of iterations, IEEE Transactions on Computers, vol. 36, pp. 850–863, 1992.

[12] A. Amin and W. Shinwari, High-radix multiplier-dividers: Theory, design, and hardware, IEEE Transactions on Computers, vol. 59, no. 8, pp. 1009–1022, August 2010.

[13] T.-J. Kwon and J. Draper, Floating-point division and square root using a taylor-series expansion algorithm, Microelectronics Journal, Available: www.elsevier.com/locate/mejo, vol. 40, pp. 1601–1605, 2009.

[14] N. Brisebarre, J.-M. Muller, and S. K. Raina, Accelerating correctly rounded floating-point division when the divisor is known in advance, IEEE Transactions on Computers, vol. 53, no. 8, pp. 1069 – 1072, August 2004.

[15] X. Wang and B. Nelson, Tradeoffs of designing floating-point division and square root on virtex fpgas, in Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on, 2003, pp. 195–203.

[16] B. P. H. Nikmehr and C. Limb, A novel implementation of radix-4 foating-point division/square-root using comparison multiples, Computers and Electrical Engineering, Available: www.elsevier.com/locate/compeleceng, vol. 36, pp. 850–863, 2010.

[17] T. Lang and A. Nannarelli, A radix-10 digit-recurrence division unit: Algorithm and architecture, IEEE Transactions on Computers, vol. 56, no. 6, pp.727 –739, June 2007.

[18] W. Liu and A. Nannarelli, Power efficient division and square root unit, IEEE Transactions on Computers, vol. 61, no. 8, pp. 1059 –1070, August 2012.

[19] K. Quinn., The newton raphson algorithm for function optimization. Department of Political Science and The Center for Statistics and the Social Sciences, pp. 364–384, October 2001.

[20] A. Y. S. Lam and V. O. K. Li, Chemical Reaction Optimization: a tutorial, 2012.

[21] T. Pham, Y. Wang, and R. Li, A variable-latency floating-point division in association with predicted quotient and fixed remainder, in Circuits and Systems (MWSCAS), 2013 IEEE 56th International Midwest Symposium on, 2013, pp. 1240–1245.

[22] A. Amaricai, M. Vladutiu, and O. Boncalo, Design issues and implementations for floating-point divide - add fused, Circuits and Systems II: Express Briefs, IEEE Transactions on, vol. 57, no. 4, pp. 295 –299, April 2010.

[23] S. Boldo and J.-M. Muller, Exact and approximated error of the fma, IEEE Transactions on Computers, vol. 60, no. 2, pp. 157 –164, February 2011.

[24] L. Huang, S. Ma, L. Shen, Z. Wang, and N. Xiao, Low-cost binary128 floating-point fma unit design with simd support, IEEE Transactions on Computers, vol. 61, no. 5, pp. 745 –751, May 2012.

[25] M. Baesler, S. Voigt, and T. Teufel, Fpga implementations of radix-10 digit recurrence fixed-point and floating-point dividers, in Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on, December 2011, pp. 13 –19.

[26] Björn Liebig, Andreas Koch, Low-Latency Double-Precision Floating-PointDivision for FPGAs, 2014 International Conference on Field-Programmable Technology (FPT), pp. 25 – 32.

MỘT CẢI TIẾN CHO SỰ ƯỚC LƯỢNG THƯƠNG SỐ CHO PHÉP TOÁN CHIA SỐ DẤU CHẤM ĐỘNG

Tóm tắt. Ngày nay, những phép tính số dấu chấm động được sử dụng như những hàm bổ trợ trong các hệ thống nhúng tư duy ứng dụng trong các lĩnh vực như vật lý, hệ thống hàng không vũ trụ, mô phỏng hạt nhân, xử lý tín hiệu hình ảnh và kỹ thuật số, hệ thống điều khiển tự động và điều khiển tối ưu và tài chính, v.v. Tuy nhiên, phép toán chia số dấu chấm động chậm hơn so với phép toán nhân số dấu chấm động. Để giải quyết vấn đề này, đã có nhiều nghiên cứu để giảm số lượng vòng lặp cần thiết để ra thương số bằng việc dùng tài nguyên bảng tra (LUT) để đạt tới xấp sỉ gần giá trị thương số. Trong bài báo này, chúng tôi đề xuất một thuật toán ước lượng cải tiến để đạt đến một thương số tối ưu bằng sự tiên đoán những bit nhất định trong số chia và số bị chia, giảm tài nguyên LUT cần thiết. Do đó, thương số cuối cùng đạt được bằng cách tích lũy tất cả các thương số tiên đoán của giải thuật tiên đoán của chúng tôi đề xuất. Kết quả thực nghiệm cho thấy chỉ cần từ 3 đến 5 vòng lặp để có được thương số cuối cùng trong phép chia số dấu chấm động. Thêm nữa, thiết kế được đề xuất chiếm 0.84% đến 3.28% (1732 LUTs đến 6798 LUTs) và 5.04% đến 10.08% (1916 (ALUT) đến 3832 (ALUT)) khi được cài đặt trên chip Xilinx Virtex-5 và Altera Stratix-III FPGAs tương ứng. Hơn nữa, thiết kế đề xuất cho phép những người sử dụng theo dõi phần dư để đặt ngưỡng tùy chỉnh cho số dư tương thích với những ứng dụng chuyên biệt của người sử dụng.

Từ khóa. Số dấu chấm động, phép toán chia số dấu chấm động, đơn vị xử lý số dấu chấm động (FPU), FPGA, bảng tra (LUT), hệ thống nhúng.

Ngày nhận bài:08/08/2019

Ngày chấp nhận đăng:25/10/2019