Single/Double Precision Floating-Point Division and Square Root Unit Based on SRT-8 Algorithm

Yuanxi Peng, Tingting He, Yuanwu Lei(✉), and Baozhou Zhu

College of Computer, National University of Defense Technology, Changsha 410073, China
yuanwulei@nudt.edu.cn

Abstract. To meet the precision requirement of different applications and reduce latency of operation for low precision, a unified structure for IEEE-754 double-precision/SIMD single-precision floating-point division and square root operation based on SRT-8 algorithm was introduced. Special instructions were designed and independent mantissa computing unit and normalization unit are implemented. Moreover, parallel adders and QDS structure was adopted to hide the latency of look-up table, generating fast addend was used to decrease critical path, and "On-the-fly" conversion was employed for saving area-cost. Experimental results show that our proposed design can achieve low latency and low hardware overhead.

Keywords: Single/double precision · SRT-8 · Division · Square root · DSP

1 Introduction

Modern applications have a wide range use of floating-point division and square root operations [1], however their precision requirement is different. For example, scientific computation requires as high as possible operational precision, while some applications, such as grayscale image processing, require lower precision computing. So, this paper researches a structure easy to implement IEEE-754 single/double precision floating-point division and square root, to satisfy the precision requirement of different applications.

Many algorithms [2] are presented to implement division and square root, which could be divided into two categories. The first category is based on multiplications, such as Newton algorithm and Goldschmidt algorithm [3], has the character of fast convergence, but their hardware structures are complicated. The other category, such as SRT [4–7], is based on addition and subtraction, and its structure is simple and easy to round with linear convergence. It's more suitable to design of flexible precision than these based on multiplication.

This work is supported by the Aerospace Science Foundation of China (No. 2013ZC88003), and the Natural Science Foundation of China (No. 61402499).

W. Xu et al. (Eds.): NCCET 2016, CCIS 666, pp. 3–14, 2016.
DOI: 10.1007/978-981-10-3159-5_1

In order to improve operation efficiency of SRT algorithm, many researchers have done many related works. In order to reduce hardware resource, [8, 9] integrated division and square root into a unit according to their similarity of implementation by reusing, but the latency of square root is obviously longer than division so that iteration frequency is low [10, 11] adopted redundant digits to express operands to eliminate the latency of addition carry, and used minimum redundant digit set to simplify generating remainder. Because the larger the redundant digit set is, the more operations are needed for generating remainder.

Obviously, there are still many challenges to SRT algorithm in the tradeoff between performance and consumption. To get a higher efficient design, this work proposed several methods. The advantages of the proposed design are as following:

(1) Independent mantissa computing and normalization structure and splitting iterative instructions were adopted to implement double-precision or SIMD (single instruction multiple data) single-precision floating-point operation. Splitting iterative instructions were designed for operating lower precisions cost less latency.
(2) Using simpler logic to implement quotient conversion on-the-fly to minimize additional required hardware.
(3) In division and square root iteration, their addend is parallel directly generating instead of computing step by step to reduce latency and improve frequency.

2 Background

SRT is a digit-recurrence algorithm to calculate division and square root. Comparing with the traditional method to compute division and square root, SRT algorithm ensures more quotient digits by the function of quotient digit selection each digit-recurrence. Traditional digit-recurrence radix is 2, 1 bit quotient digit is produced each iteration. This work researched SRT algorithm with radix 8(SRT-8), and 3-bit quotient digits are produced at each iteration.

The operation processing is similar for division and square root in SRT-8 algorithm. Equation (1) is consolidated to iteration for division and square root.

$$W[j+1] = 8 \times W[j] + F[j]. \tag{1}$$

In Eq. (1), $W[j]$ denotes the remainder after jth iteration, and $F[j]$ is the addend for iteration. As for division, $F[j]$ equals to $-d \times q_{j+1}$ where d is divisor, and q_{j+1} is the quotient digit and selected according to $W[j]$ and divisor. The division iterative can be written as Eq. (2):

$$W[j+1] = 8 \times W[j] - d \times q_{j+1}. \tag{2}$$

For square root, $F[j]$ equals to $-2Q[j]q_{j+1} - q_{j+1}^2 8^{-(j+1)}$, where $Q[j]$ is the quotient after jth iteration, and q_{j+1} is the quotient digits and selected according to $W[j]$ and $Q[j] = \sum_{i=1}^{k} q_i 8^{-i}$. The square root iterative can be written as Eq. (3).

$$W[j+1] = 8 \times W[j] - 2Q[j]q_{j+1} - q_{j+1}^2 8^{-(j+1)}. \quad (3)$$

The division or square root result can be reach to target precision by sequential addition and shifting.

3 Structure Supporting Single/Double Precision Operation

In this section, the structure of double/SIMD single precision floating-point unit for division and square root based on SRT-8 algorithm is proposed. Special instructions and novel scheduling process were designed to implement division and square root operation.

3.1 Structure of Independent Mantissa Computation and Normalization

Because of the similarities in the digit-recurrence algorithm, division and square root can be integrated into the same unit. The structures of independent mantissa computing unit and normalization unit are designed for separating division or square root mantissa operation and normalization, as shown in Fig. 1. The independent structure is easy to execute flexible precision floating-point operation joined with corresponding splitting iterative instructions scheduling. Its operation process is sending the operands to mantissa computing unit for computing mantissa quotient or square root firstly, then, sending source operands and mantissa operation result to normalization unit for final processing. Double/SIMD single precision operation is implemented by changing times of instructions scheduling.

Each iteration of division or square root can be completed within one cycle for mantissa computing unit. Its iteration begins with SRT-8 core operation, then, storing the quotient and residue result, and according to the result preparing data and latching data for the next iteration. The data prepared has two different cases due to whether it is the first iteration. For the first one, the iteration data is obtained from source operands, otherwise, the iteration data is obtained from the last iteration results.

To enhance instruction level parallel, the proposed unit supports SIMD operation by two single-precision data path as show in Fig. 2. Two quotient digit selection (QDS) modules are set in mantissa computing unit. When operand's format is double-precision floating-point, QDS(H) will work, otherwise, the operand's format is double single-precision, QDS(H) and QDS(L) will work at same time corresponding to high-order single-precision operand and low-order single-precision operand. Double-precision format data and double single-precision format data adopted same 57 bits data path. Normalization unit adopted double path structure supporting double-precision and single-precision format floating-point operations.

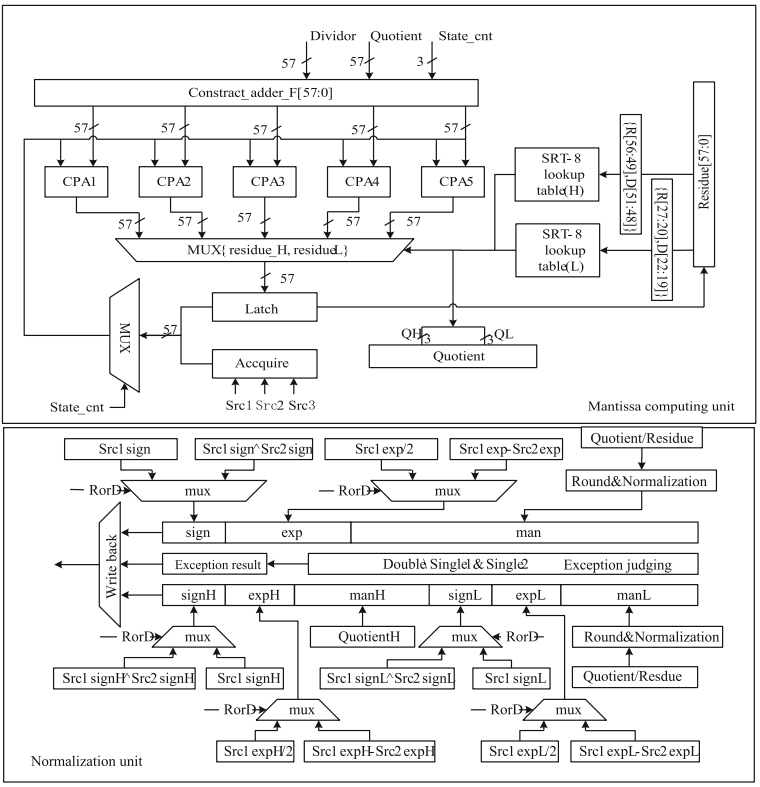

Fig. 1. Structures of independent mantissa computing unit and normalization unit

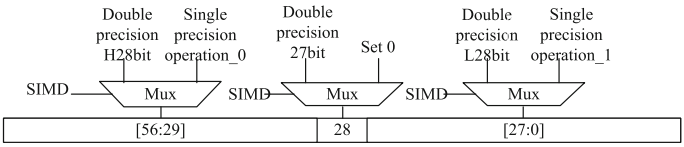

Fig. 2. SIMD data path for single-precision operation

3.2 Splitting Iteration Instructions

According to IEEE-754 floating-point standard, the mantissas of single-precision and double-precision floating-point are 24 bits and 53 bits, respectively. The proposed mantissa computing unit based on SRT-8 algorithm generates 3 bits quotient digits for each iteration. Then, 8 iterations and 18 iterations are needed for single-precision operation and double-precision operation, respectively. However, some operations need lower precision, they can be processed with less iterations.

Splitting iteration instructions are easy to achieve flexible precision floating-point division and square root operations through different times of execution. In this section,

7 cycles for double precision and 4 cycle for single precision division and square root splitting iteration instructions were proposed. They can support different precision floating-point operations by multiple and combined scheduling instructions.

We designed SRT-8 instructions which were instructions (FSRTDD and FSRTDS) for supporting division SRT iteration and instructions (FSRTRD and FSRTRS) for supporting square root SRT-8 iteration. At the same time, normalization instructions (FNORMD and FNORMS) are designed to final normalization processing, which executes one cycle.

The double-precision SRT-8 instructions FSRTDD and FSRTRD designed support six iterations for division and square root, and 18 bits quotient digits are produced each scheduling instruction, while the SIMD single-precision SRT-8 instructions FSRTDS and FSRTRS designed support three iterations for division and square root, and 9 bits quotient digits are produced each scheduling instruction. How many times are needed to schedule depend on target operation precision. Once the enough precision result is obtained, the source operands and the mantissa result will be sent to normalization unit and operated by the normalization instruction corresponding to target precision data format. When floating-point division or square root is executed, the SRT-8 instruction need be scheduled three times for single precision or double precision. If operation is lower precision, flexible precision result matching single or double FP format can be achieved by changing instruction executing times.

Because of each instruction scheduling supports six or three iterations, the intermediate result need be stored in appointed register files for the next scheduling, which includes partial quotient, residue and some scheduling information. In the first scheduling, its source operands are dividend and divisor, while its source operands are residue, divisor and the third operand for other scheduling. The third operand comes from destination operand stored in the last scheduling, which is composed of partial quotient digits and scheduling times.

The following displayed is the scheduling process for two kinds of precision division. Because of the scheduling process of instruction FSRTRD and FSRTRS for square root are similar to the division instruction, its description is omitted (Tables 1 and 2).

Table 1. Instruction scheduling of single-precision division

Cycle	Instruction
1	FSRTDS R1, R2, R3, R5:R4;
2–4	SNOP;
5	FSRTDS R5, R2, R4, R7:R6;
6–8	SNOP;
9	FSRTDS R7, R2, R6, R9:R8;
10–12	SNOP;
13	FNORMS R1, R2, R9, R10;

Table 2. Instruction scheduling of double-precision division

Cycle	Instruction
1	FSRTDD R1, R2, R3, R5:R4;
2–7	SNOP;
8	FSRTDD R5, R2, R4, R7:R6;
9–14	SNOP;
15	FSRTDD R7, R2, R6, R9:R8;
16–21	SNOP;
22	FNORMD R1, R2, R9, R10;

The instruction FSRTRD and FSRTRS need three source operands. In the first scheduling for division, source operand 1 is dividend, source operand 2 is divisor, and source operand 3 is zero. In the follow scheduling, source operand 1 is remainder, source operand 2 is still divisor, and source operand 3 is composed of quotient digits and recording of iteration scheduling. At the same time, source operand 1 and source operand 3 are destination operands from last scheduling. The instructions FNORMD and FNORMS are used for normalization. The instruction SNOP inserted is used to wait for iteration processing.

4 Implementation of Unified Unit for Division and Square Root

In this section, the implementation of mantissa computing unit is introduced based on SRT-8 algorithm.

4.1 Quotient Digit Selection

Quotient Digit Selection (QDS) is significant part of SRT algorithm. The leading digits of residual and divisor are as inputs for the QDS function, and the quotient digits are produced corresponding to current iteration.

For constructing the QDS function, some issues need to be solved, including by the following:

1. The range area of quotient digit selection that is quotient set;
2. How many bits of remainder and divisor the input of the function;
3. How to map quotient from remainder joined on divisor;

A symmetrical redundancy quotient set is used for high speed operation. That is:

$$q = \mathrm{k} \in \{\bar{a}, \bar{a}+1 \cdots, -1, 0, 1, \cdots, a-1, a\}.$$

The set determines the redundancy factor ρ, and the factor ρ is defined by $\rho = \frac{a}{8-1}$. And $1 > \rho > \frac{1}{2}$ so, parameter 'a' belongs to (5, 6, 7) for SRT-8.

$$\begin{cases} P_{\min} = L_k = (-\rho + k)d \\ P_{\max} = U_k = (\rho + k)d \end{cases}. \tag{4}$$

In Eq. (4), L_k and U_k are the low bound and the up bound of remainder when quotient is k, and $U_{k-1} > L_k$. $\{-5, 5\}$ is selected as the range area of quotient.

According to Eq. (4) and the confirmed quotient set, the range of remainder corresponding to each quotient can be obtained, as shown in Table 3. Among the overlap area, the larger quotient digit is selected. Figure 4 is P-D diagram for digit selection. The largest value of remainder is $\frac{40}{7}d$, so the bits of input $W[j]$ are larger 3 than input d at least for meeting the range of remainder.

Table 3. W[j] bounds/7

Quotient q	0	1	2	3	4	5
L_k (d)	−5d	2d	9d	16d	23d	30d
U_k (d)	5d	12d	19d	26d	33d	40d

The divisor $d \in [1, 2)$ is divided into smaller interval $[d_i, d_{i+1})$ whose length is $2^{-\delta}$, and $d_1 = 1/2$, $d_{i+1} = d_i + 2^{-\delta}$. Then, the leading δ bits of divisor are used to represent the approximation of divisor. When the internal is in $[d_i, d_{i+1})$ and q_{j+1} equals to k, the residual belongs to $m_k(i) \leq 8Q_j < m_{k+1}(i)$, m_k needs meet the two constrains as following:

$$\begin{cases} m_k(i) \geq \max\{L_k(d_i), L_k(d_{i+1})\} \\ m_k(i) \leq \min\{U_{k-1}(d_i), U_{k-1}(d_{i+1})\} \end{cases}$$

The minimum length of the selection constant is arranged C bits, then $m_k(i) = A_k(i)2^{-C}$, $A_k(i)$ is integer.

$$\begin{cases} L_k\left(d_i + 2^{-\delta}\right) \leq A_k(i)2^{-C} \leq U_{k-1}(d_i), Q[j] \geq 0 \\ L_k(d_i) \leq A_k(i)2^{-C} \leq U_{k-1}\left(d_i + 2^{-\delta}\right), Q[j] < 0 \end{cases}.$$

The worst-case k = 5, d = 1 it derives

$$\delta \geq \log_2 \frac{a - \rho}{(2\rho - 1)d_{\min}} = \lceil \log_2 10 \rceil = 4.$$

The minimum value of δ is 4 amount to input d need 4 bits. According the above analysis, the input W[j] is larger 3 than d, so the minimum value of C is 7.

Square root and division can use the same quotient selection, which is proofed in [4] in detail.

4.2 Parallel SRT-8 Iteration Unit

As shown in Fig. 3, the iteration unit includes five adders corresponding to five absolutes of possible quotient in quotient digit set, and a quotient digit selection (QDS) modules for generating quotient digits each iteration. Addition and QDS can be parallel execution, then, the next residue is selected from five addition results depend on the quotient digits selected.

4.3 Parallel Generating Addend *F*

While QDS function is working, all possible cases for addend F is generated concurrently, in order to hide the latency of quotient selection, and the all F are sent to five parallel adders.

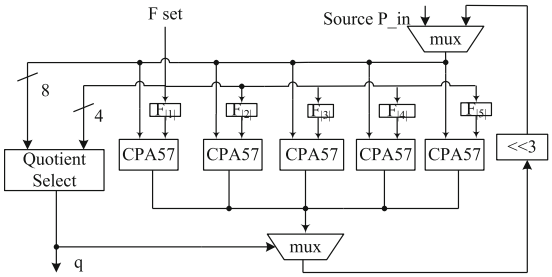

Fig. 3. SRT-8 iteration unit

In division, the input of adder F is simple. The $F = qd$ includes five cases: d, 2d, 3d, 4d and 5d. All of them, d is divisor. 2d and 4d can be obtained by d shifted. 3d is the sum of d and 2d. 5d is the sum d and 4d. The F sets generated are sent to five adders to operate $W[j+1] = W[j] \times 8 \pm F[j]$ for producing next remainder.

In square root, Q and QM representing positive quotient and negative quotient, the expression of F[j] is converted to the following:

$$F[j] = \begin{cases} -2Q[j]q_{j+1} - q_{j+1}^2 8^{-(j+1)}, q_{j+1} \geq 0 \\ 2QM[j]\left|q_{j+1}\right| + \left(2 \times 8 - \left|q_{j+1}\right|\right)\left|q_{j+1}\right|8^{-(j+1)}, q_{j+1} < 0 \end{cases}$$

The generating process of F is complex from function computing step by step, but also the process needs longer latency. In order to reduce the latency for generating F, the design adopted direct look-up table to produce F corresponding to each quotient digits. The string 'a..aa' and 'b..bb' replace the value of $Q[j]$ and $QM[j]$ respectively. Then, the updates of F are as shown in Table 4.

Table 4. Generating F[j] directly

q_{j+1}	Expression	F[j] item
1	$-2Q[j] - 8^{-(j+1)}$	a..aa0001
−1	$2QM[j] + 15 * 8^{-(j+1)}$	b..bb1111
2	$-4Q[j] - 4 * 8^{-(j+1)}$	a..a00100
−2	$4QM[j] + 12 * 8^{-(j+1)}$	b..b11000
3	$-2Q[j] - 4Q[j] - 9 * 8^{-(j+1)}$	a..aa000110 + a..aa0000
−3	$2QM[j] + 4QM[j] + 39 * 8^{-(j+1)}$	b..bb0101001 + b..bb0000
…	…	…

As shown in Fig. 4, 10 possible F sets are generated corresponding to 10 different quotient digits simultaneously based on values assigned in advance. The right F are chosen by the logic of multi-Choice, Count_iteration (the signal for recording iterations), $Q[j]$, $QM[j]$ as input index.

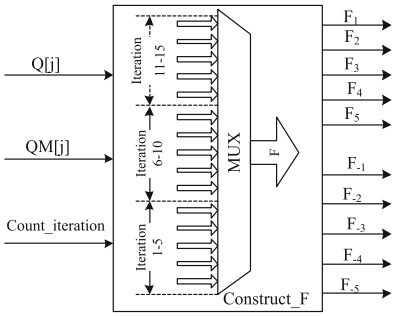

Fig. 4. Parallel generating addend F

4.4 Quotient Conversion "On-the-Fly"

In SRT-8 algorithm, the quotient is typically collected in a representation where the digits can take on both positive and negative values. Thus, at some point, all of the values must be combined and converted into a standard representation. This requires a full-width addition for the conversion, which can be a slow operation. Techniques exist for performing this conversion "on-the-fly", therefore the extra cycle may not be needed [13]. But if this scheme is complex, it will add more required hardware. Then, this conversion "on-the-fly" may bring the problem that the penalty for requiring the additional cycle is obviously much larger than the benefit from it. Focused on the issue, we proposed a conversion "on-the-fly" with little additional required hardware for controlling logic. The follow equations are the definition of positive quotient and negative quotient respectively.

$$Q[j] = \sum_{i=1}^{k} q_i r^{-i} \text{ and } QM[j] = Q[j] - r^{-j}$$

As shown in Fig. 5, the implementation in [12] of the algorithm uses two registers to store Q[j] and QM[j].

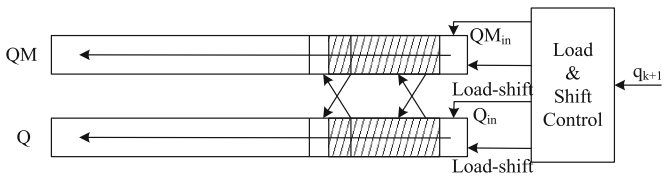

Fig. 5. Traditional quotient conversion on-the-fly

These registers can be shifted three digits left with insertion in the least significant position depend on q_{j+1}. They also require parallel loading to replace Q[j] with QM [j + 1] and vice versa.

The proposed conversion only uses one register to complete on the fly. As shown in Fig. 6, register Q was set to restore quotient and register E_QM_{in} to restore negative quotient digits corresponding to each q_j. $Neg\_q = 7 - q_j$ for $q < 0$; $Neg\_q = q_{j-1}$ for $q > 0$. The next quotient digits are restored in register Q by shifting right. If positive quotient is selected, the quotient digital q_{j+1} is restored directly, otherwise, the last restored q_j will be replaced the last restored E_QM_{in}, and q's complement will be restored as the updating quotient digits.

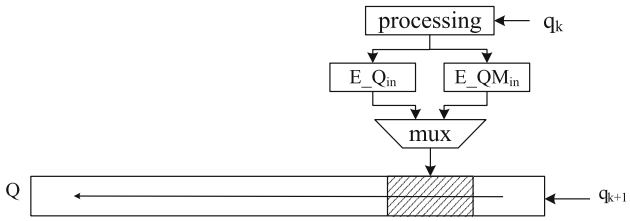

Fig. 6. Quotient conversion on-the-y with simple logic

5 Experiments

We have implemented the proposed single/double precision floating-point division and square root unit and instructions on X-DSP. All modules were encoded in Verilog and synthesized with 45 nm technology library in typical conditions (1 V, 25 °C), and clock cycle was set as 450 ps. At the same time, the other possible designs were implemented as comparisons which were iteration unit for division and square root respectively.

As shown in Table 5, experiment results show that unified unit for division and square root need only be increased by small area cost upon individual SRT-8 division or square root units.

Table 5. Results from synthesis

Operation unit	Area (μm^2)	Power	Critical
Division operation unit	12222.63	10.66	427
Square root operation unit	16039.41	12.48	441
Unified operation unit	18589.36	14.10	446

The implemented unit is capable of calculating single precision a double precision floating-point data format division and square root. Table 6 describes cycles and latency according to different target precision operations need.

Error analyze, the precision of divider was verified by recursion division. The proposed design and C program respectively operated recursion division using same data. Then, compare their results. C program was run on Intel processor. The experimental results show the design can obtain the precision of floating-point division same with Intel processor.

Table 6. Performance of different target precisions

Data format precision	Cycles	Latency (ns)
Single precision 9-bit	5	2.25
Single precision 18-bit	9	4.05
Single precision 24-bit	15	6.75
Double precision 18-bit	8	3.60
Double precision 32-bit	15	6.75
Double precision 53-bit	22	9.90

This paper proposed improved implementation of on-the-fly conversion. The design without on-the-fly conversion, the design with on-the-fly conversion proposed by [12] and the design with on-the-fly conversion proposed by above section were implemented respectively and synthesized in same experimental environment. The experimental data was shown in Table 7 taking double precision operation as example.

Table 7. Comparison of different structures

Structure	Area	Power	Cycles	Latency (ns)
Design without on-the-fly	17925.22	12.64	24	13.50
[13] 's design	19707.41	15.39	22	12.60
Our design	18589.36	14.10	22	12.60

Obviously, the design with on-the-fly conversion cost more area than the design without on-the-fly conversion, but latency is decreased by the structure for computing. In addition, our design area-cost is decreased by 6% comparing to the [13]'s design.

In previous works [13–15], generation of F in square root was implemented by complex computing process. Comparing with directly parallel generating F proposed, previous design will enlarge the latency of critical path, when division and square root were implemented in unify hardware structure, because generation of F in square root need larger latency than division by computing. For analyzing the contribution of design latency, we implemented design unit with past method of generating F for comparing. The experimental data is shown that its area is 15839.21 μm^2 and is decreased by 15% comparing proposed design, however, its critical path is 570 ps and is longer than the proposed design.

6 Conclusions

This work presents the design and implementation of single/double precision floating-point division and square root unit based on SRT-8 algorithm. Structure of independent mantissa computing and normalization joined with splitting iteration instructions are adopted for implementing flexible precision floating-point operation. With the above mentioned improved and optimized strategies, the design obtained higher performance than others. Especially for low precision operation, make them cost less latency.

References

1. Oberman, S.F., Flynn, M.J.: Design issues in division and other floating-point operations. J. IEEE Trans. Comput. **46**(2), 154–161 (1997)
2. Inwook, K., Earl, E.S.: A Goldschmidt division method with faster than quadratic convergence. IEEE Trans. Very Large Scale Integr. Syst. **19**(4), 759–763 (2011)
3. Stuart, F.O., Michael, J.F.: Division algorithms and implementations. IEEE Trans. Comput. **46**(8), 833–854 (1997)
4. Peter, K.: Digit selection for SRT division and square root. IEEE Trans. Comput. **54**(3), 727–739 (2005)
5. Dong, W., Milobs, D.E.: A Radix-16 combined complex division/square root unit with operand prescaling. IEEE Trans. Comput. **61**(9), 1243–1255 (2012)
6. Ingo, R., Tobias, G.N.: Digit-set-interleaved Radix-8 division/square root Kernel for double-precision floating point. In: 2010 International Symposium on System on Chip (SoC), Tampere, Finland, pp. 150–153 (2010)
7. Ercegovac, M.D., Lang, T.: Division and Square Root: Digit Recurrence Algorithms and Implementations. Kluwer Academic Publishers, Norwell (1994)
8. Frandrianto, J.: Algorithm for high-speed shared Radix-8 division and Radix-8 square root. In: Proceedings of 9th Symposium on Computer Arithmetic, pp. 68–75 (1989)
9. Nannarelli, A.: Radix-16 combined division and square root unit. In: 2011 20th IEEE Symposium on Computer Arithmetic, pp. 169–176 (2011)
10. Amaricai, A., Boncalo, O.: SRT Radix-2 dividers with (5, 4) redundant representation of partial remainder. IEEE Trans. 1016–1020 (2013)
11. Issad, M., Anane, M., Bessalah, H.: Influence de la Base sur les Performance de la Division SRT. Journes Francophones sur Adquation algorithm architecture 91–94 (2005)
12. Ercegovac, M.D., Lang, T., Milo, D.: On-the-fly rounding. IEEE Trans. Comput. **41**(12), 1497–1503 (1992)
13. Nannarelli, A.: Radix-16 combined division and square root unit. In: 2011 20th IEEE Symposium on Computer Arithmetic, Germany, pp. 169–176 (2011)
14. Ingo, R., 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 (SoC), Tampere, Finland, pp. 150–153 (2010)
15. Wetter, H., Schwarz, E.M., Haess, J.: The IBM eServer z990 floating-point unit. IBM J. Res. Dev. **48**(3), 311–322 (2004)