2023 IEEE 30th Symposium on Computer Arithmetic (ARITH) | 979-8-3503-1922-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ARITH58626.2023.00013

Dual-Purpose Hardware Algorithms and Architectures – Part 1: Floating-Point Division

Jihee Seo[1,2] and Dae Hyun Kim[2]
[1]Synopsys, Inc., Hillsboro, OR, USA
[2]School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, USA
jiheeseo@synopsys.com, daehyun.kim@wsu.edu

***Abstract*—Division is a time-consuming, but frequently-used arithmetic operation, so an enormous amount of effort has been made to improve the performance of dividers. Most of the division algorithms in the literature are offline algorithms that minimize the execution time of a single division, whereas some others are online algorithms that maximize the throughput (# divisions executed per time). In this paper, we propose an interval-analysis-based normal-binary floating-point division algorithm that can be used for both offline and online division. We implement two offline and four online dividers using the algorithm and compare them with recently-proposed offline and online dividers. The simulation results show that the offline versions are the best for a Binary64 offline division, whereas the online versions are the best for a Binary64 online division.**

***Index Terms*—Divider; Floating-Point Arithmetic; Online Division;**

I. Introduction

Division is one of the most commonly-used arithmetic operations, but it is often a serious performance bottleneck [1], [2]. Therefore, many researchers have put a great amount of effort in the design of high-performance dividers [3]. Meanwhile, computer architectures have been improved for both high-speed and high-throughput computing. A *high-speed* design minimizes the execution time of a single operation, whereas a *high-throughput* design maximizes the throughput (the number of operations executed per unit time) [4]. Most of the division algorithms aim for high-speed division [5]–[9].

Pipelining of an arithmetic unit for an operation splits the logic into n stages so that several independent operations of the same type can be executed sequentially in the pipeline stages. In this case, the throughput increases roughly by $n\times$. Thus, pipelining is a representative technique to increase the throughput at the cost of additional flip-flops. Unfortunately, pipelining cannot improve the throughput when operations are dependent on each other. For example, $(a/b)/c$ is decomposed into $i_1 : x = a/b$ and $i_2 : y = x/c$ where the dividend of i_2 is the result of i_1, so i_2 is dependent on i_1. In this case, the execution of i_2 has to wait until i_1 has finished. Although some architecture-level techniques such as out-of-order execution can help resolve the dependencies, they cannot completely resolve all the dependencies among arithmetic operations.

Online algorithms enable the parallel execution of i_1 and i_2 by passing partial results of i_1 to the arithmetic unit executing i_2 and utilizing them for the computation of i_2, thereby reducing the overall execution time and improving the throughput. Thus, several online division algorithms have also been proposed in the literature [10]–[16].

In this paper, we present a normal-binary floating-point division algorithm that can be used for both offline and online (*dual-purpose*) floating-point division. A special case of the algorithm is the normal-binary floating-point offline division in [17]. Our contributions in this paper are as follows:

- The division algorithm we propose uses the conventional non-redundant binary (normal binary) number system, so it does not require hardware for the conversion between redundant and non-redundant binary number systems.
- The algorithm can be used for both offline and online division.
- The algorithm fully utilizes all the given dividend bits. In addition, if the divisor of a division is fully given at the beginning of the division, the division algorithm fully utilizes the divisor. These two features help reduce the execution time significantly.
- The algorithm works on any number of dividend and divisor bits given.

The rest of this paper is organized as follows. We show the motivation of this paper and review online and digit-recurrence division in Section II. In Section III, we review the normal-binary floating-point offline division and derive it using interval analysis. In Section IV, we present normal-binary floating-point online division using interval analysis. Then, we show hardware implementation of the algorithm in Section V. In Section VI, we compare several offline and online dividers, then we conclude in Section VII.

II. Preliminaries

In this section, we show the motivation of this paper and review online and digit-recurrence division.

A. Motivation

Data dependencies among arithmetic operations occur frequently in many applications. For example, the following code is used in GNU Scientific Library (GSL):

$$\sqrt{\frac{a}{b}} \cdot (c \cdot \cos\, x + d \cdot e \cdot \sin\, y). \tag{1}$$

For $\sqrt{a/b}$ in the code, we should compute a/b before computing its square root. In this case, online square root algorithms [18] can reduce the overall execution time of $\sqrt{a/b}$.

979-8-3503-1922-4/23/$31.00 ©2023 IEEE
DOI 10.1109/ARITH58626.2023.00013

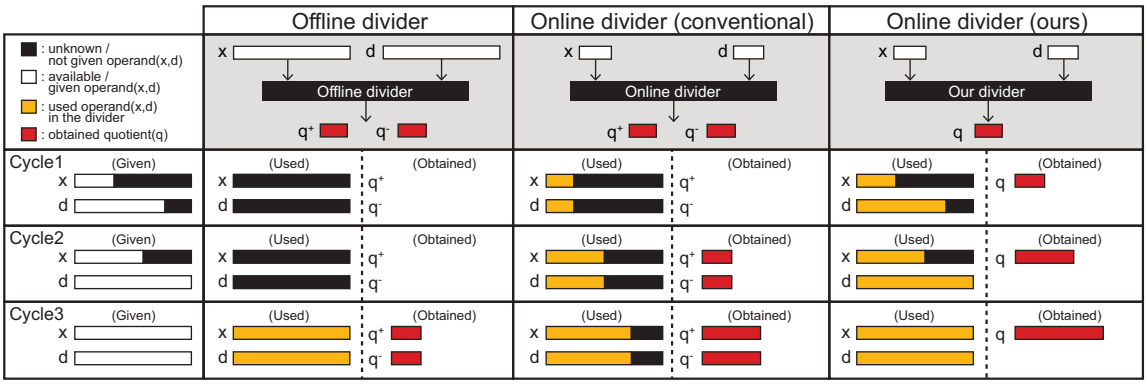

Fig. 1. An overview of the operations of offline, online, and the proposed dividers for an online division.

TABLE I
EXECUTION TIME REDUCTION BY ONLINE ARITHMETIC

Benchmark	MM	DM	MD	DD	Total
FEM	0%	0.58%	3.85%	0.36%	4.79%
EM simulation	5.71%	0%	0%	0%	5.71%
Statistical inference	0.56%	0.25%	0.33%	0.43%	1.57%
Bessel function	23.61%	0%	1.47%	2.96%	28.04%

For a quantitative analysis of the effectiveness of online arithmetic, we used a cycle-accurate architecture simulator with several benchmarks to simulate online arithmetic. In this analysis, we focused on four data dependencies as follows:

- Multiply-multiply (MM): $(a \times b) \times c$ or $a \times (b \times c)$
- Multiply-divide (MD): $(a \times b)/c$ and $a/(b \times c)$
- Divide-multiply (DM): $(a/b) \times c$ or $a \times (b/c)$
- Divide-divide (DD): $(a/b)/c$ and $a/(b/c)$

In this analysis, we assumed ideal online multipliers and dividers that can (1) start their computations two cycles after the first digits of the dependent operands are received and (2) generate one digit every clock cycle from that.

Table I shows the execution time reduction by each online arithmetic. For example, the use of online dividers that can resolve the multiply-divide (MD) dependency reduces the execution times of the four benchmarks by 3.85%, 0%, 0.33%, and 1.47%, respectively. The EM simulation does not benefit from it because it does not have MD dependencies among the instructions. The use of online dividers that can resolve the DD dependency reduces the execution time of the Bessel function benchmark by 2.96%.

B. Online Division

As explained in Section I, several online division algorithms have been proposed in the literature [10]–[16] and most of them use digit-recurrence division algorithms. For example, the online division algorithm proposed in [10] obtains a new digit of the quotient of a division using a selection function based on the digit-recurrence algorithm upon receiving an additional digit of each operand. However, it should obtain a quotient digit from incomplete operands while guaranteeing the convergence of the quotient. Trivedi [11] extended [10] to a radix-4 online division algorithm. Tenca [16] modified the online division algorithm presented in [19] and improved the performance of the algorithm at the cost of increased area.

In this paper, we propose a dual-purpose division algorithm that can be used for both offline and online division. If the dividend and the divisor of a division are offline (fully given at time 0), the division algorithm works as an offline algorithm, i.e., it utilizes all the given bits. If one or both of the operands are online, then the algorithm works as an online algorithm and tries to find quotient bits from the partially given operands.

Fig. 1 shows an overview of the operations of offline and online dividers for an online division. The operands are given in three clock cycles (shown in the leftmost column). An offline divider needs complete operands, so it waits for two clock cycles without any computation. On the other hand, an online divider starts the division with the incomplete operands in the first cycle and obtain some quotient digits. Notice that online dividers generally use redundant number systems (q^+ and q^- in the figure). In the second and third cycles, it uses more operand digits and obtains more quotient digits. In the proposed divider, we use all the dividend and divisor digits given at the first cycle and obtain quotient digits, which is the biggest difference between the proposed and other online dividers. Notice also that our divider uses the normal binary number system.

C. Digit-Recurrence Floating-Point Division

For $x = d \cdot q + rem$, radix-r digit-recurrence division algorithms use the condition $|rem| < |d| \cdot ulp$ where ulp is the unit in the last place (r^{-n} for $q = \sum_{i=0}^{n} q_i \cdot r^{-i}$). This condition is converted into the following recurrence formula

$$w[j+1] = r \cdot w[j] - d \cdot q_{j+1} \tag{2}$$

and a digit-selection function $SEL(w[j], d)$ [19]. Each digit q_i is one of $D_a = \{-a, -a+1, \cdots a-1, a\}$ for a certain a and the algorithms use redundant binary number systems for the division. Many digit-recurrence division algorithms are based on this basic division framework [5]–[7], [9].

III. Normal-Binary Offline Division

In this section, we review the floating-point offline division algorithm using the normal binary number system [19].

A. Notations and Assumptions

In the division $x = d \cdot q + rem$, x, d, and q are the dividend, the divisor, and the quotient, respectively. The IEEE standard for floating-point arithmetic [20] uses a separate bit for the sign of an operand and assumes normalized operands. Thus, we assume that $x, d \in [1, 2)$. As a result, the division above should satisfy the following condition:

$$0 \le rem < d \cdot ulp. \tag{3}$$

The quotient should also be rounded and normalized after division. The quotient obtained up to the j-th digit is written as follows:

$$q[j] = q_0.q_1 \cdots q_j = \sum_{i=0}^{j} q_i \cdot r^{-i}. \tag{4}$$

The final quotient before rounding and normalization is $q[n]$.

B. Normal-Binary Floating-Point Offline Division

In this paper, we use the normal binary system, so $q_i \in \{0, 1, \cdots, r-1\}$. For $x = d \cdot q + rem$, the followings should be satisfied for each j [19]:

$$\begin{aligned} 0 \le rem[j] < d \cdot r^{-j} &\Leftrightarrow 0 \le x - d \cdot q[j] < d \cdot r^{-j} \\ &\Leftrightarrow 0 \le r^j \cdot (x - d \cdot q[j]) < d. \end{aligned} \tag{5}$$

Similarly, $q[j+1]$ should satisfy the following:

$$\begin{aligned} & 0 \le r^{j+1} \cdot (x - d \cdot q[j+1]) < d \\ \Leftrightarrow\ & 0 \le r^{j+1} \cdot \{x - d \cdot (q[j] + q_{j+1} \cdot r^{-(j+1)})\} < d \\ \Leftrightarrow\ & 0 \le r^{j+1} \cdot (x - d \cdot q[j]) - d \cdot q_{j+1} < d. \end{aligned} \tag{6}$$

Define $w[j]$ as follows:

$$w[j] = r^j \cdot (x - d \cdot q[j]). \tag{7}$$

Then $q_{j+1} = k$ if the following is satisfied:

$$\begin{aligned} & 0 \le r^{j+1} \cdot (x - d \cdot q[j]) - k \cdot d < d. \\ \Leftrightarrow\ & k \cdot d \le r^{j+1} \cdot (x - d \cdot q[j]) < (k+1) \cdot d. \quad (8) \\ \Leftrightarrow\ & k \cdot d \le r \cdot w[j] < (k+1) \cdot d. \quad (9) \end{aligned}$$

Although this requires two inequalities, finding q_{j+1} requires the evaluation of only $r - 1$ inequalities as follows:

$$\begin{aligned} 0 &\le r \cdot w[j] - 1 \cdot d, \\ 0 &\le r \cdot w[j] - 2 \cdot d, \\ &\cdots \\ 0 &\le r \cdot w[j] - (r-1) \cdot d. \end{aligned} \tag{10}$$

Suppose $f_k = 1$ (or 0) if $0 \le r \cdot w[j] - k \cdot d$ is true (or false). Then, the bit string $F = (f_1 f_2 \cdots f_{r-1})$ is one of $(000\cdots0)$, $(100\cdots0)$, $(110\cdots0)$, $(111\cdots0)$, or $(111\cdots1)$. Thus, we can find q_{j+1} by evaluating the above inequalities and F. After we find q_{j+1}, we compute $w[j+1] = r \cdot w[j] - d \cdot q_{j+1}$.

The number of inequalities to evaluate goes up as r increases, so r should be sufficiently small. We find that $r = 2$, 4, 8, and 16 are practical radices for the division.

C. Example: Radix-4 Offline Division

Suppose $n = 4$, $r = 4$, $x = 1.2002_4$, and $d = 1.0112_4$. Since $x \ge d$, $q_0 = 1$ and $w[0] = 0.1230_4$.

Applying (10) leads to the following:

$$\begin{aligned} f_1 &: 0 \le 4 \cdot 0.1230_4 - 1 \cdot 1.0112_4 \text{ (True)}, \\ f_2 &: 0 \le 4 \cdot 0.1230_4 - 2 \cdot 1.0112_4 \text{ (False)}, \\ f_3 &: 0 \le 4 \cdot 0.1230_4 - 3 \cdot 1.0112_4 \text{ (False)}. \end{aligned}$$

The bit string $F = (f_1 f_2 f_3)$ is (100), so $q_1 = 1$. $w[1] = 0.2122_4$. Applying (10) results in the following:

$$\begin{aligned} f_1 &: 0 \le 4 \cdot 0.2122_4 - 1 \cdot 1.0112_4 \text{ (True)}, \\ f_2 &: 0 \le 4 \cdot 0.2122_4 - 2 \cdot 1.0112_4 \text{ (True)}, \\ f_3 &: 0 \le 4 \cdot 0.2122_4 - 3 \cdot 1.0112_4 \text{ (False)}. \end{aligned}$$

$F = (110)$, so $q_2 = 2$. $w[2] = 0.0330_4$. For q_3:

$$\begin{aligned} f_1 &: 0 \le 4 \cdot 0.0330_4 - 1 \cdot 1.0112_4 \text{ (False)}, \\ f_2 &: 0 \le 4 \cdot 0.0330_4 - 2 \cdot 1.0112_4 \text{ (False)}, \\ f_3 &: 0 \le 4 \cdot 0.0330_4 - 3 \cdot 1.0112_4 \text{ (False)}. \end{aligned}$$

$F = (000)$, so $q_3 = 0$. $w[3] = 0.3300_4$. For q_4:

$$\begin{aligned} f_1 &: 0 \le 4 \cdot 0.3300_4 - 1 \cdot 1.0112_4 \text{(True)}, \\ f_2 &: 0 \le 4 \cdot 0.3300_4 - 2 \cdot 1.0112_4 \text{(True)}, \\ f_3 &: 0 \le 4 \cdot 0.3300_4 - 3 \cdot 1.0112_4 \text{(True)}. \end{aligned}$$

$F = (111)$, so $q_4 = 3$. $q = 1.1203_4$ and $rem = x - d \cdot q = 0.00001332_4 < d \cdot ulp = 0.00012002_4$, so it satisfies (3).

D. Interval-Analysis-Based Offline Division

The digit-selection function in the previous section is derived from the definition of division (3). We can also derive the digit-selection function from interval analysis as follows.

For $x = d \cdot q + rem$, suppose we have obtained $q[j] = q_0.q_1 \cdots q_j$. This means that x/d satisfies the following:

$$q[j] \le \frac{x}{d} < q[j] + r^{-j}. \tag{11}$$

In this case, $q_{j+1} = k$ if the following condition is satisfied:

$$q[j] + k \cdot r^{-(j+1)} \le \frac{x}{d} < q[j] + (k+1) \cdot r^{-(j+1)}, \tag{12}$$

which is shown in Fig. 2. Rearranging the terms in (12) leads to (8), so (12) is exactly the same as the digit-recurrence division algorithm. An advantage of the derivation of (12) using interval analysis is that we can use it for online division as shown in the next section.

IV. Normal-Binary Online Division

A. Normal-Binary Online Division Algorithm

For $x = d \cdot q + rem$, suppose x and d are partially given until iteration j and we have obtained q from the partially-given x and d until iteration j as follows:

$$x[j] = x_0.x_1 \cdots x_{a[j]} = \sum_{i=0}^{a[j]} x_i \cdot r^{-i}, \tag{13}$$

$$d[j] = d_0.d_1 \cdots d_{b[j]} = \sum_{i=0}^{b[j]} d_i \cdot r^{-i}, \tag{14}$$

$$q[j] = q_0.q_1 \cdots q_{c[j]} = \sum_{i=0}^{c[j]} q_i \cdot r^{-i}, \tag{15}$$

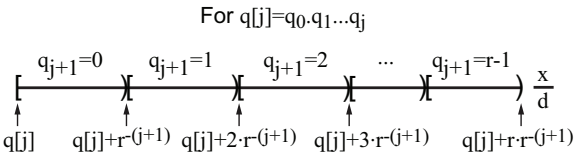

Fig. 2. Interval-analysis-based offline division.

where $a[j]$ and $b[j]$ are the indices of the rightmost digits of x and d, respectively, given until iteration j, and $c[j]$ is the index of the rightmost digit of q obtained until iteration j. In this case, the range of x/d is as follows:

$$\frac{x}{d} = \left[\left(\frac{x}{d}\right)_{j,\mathrm{MIN}}, \left(\frac{x}{d}\right)_{j,\mathrm{MAX}}\right], \tag{16}$$

$$\left(\frac{x}{d}\right)_{j,\mathrm{MIN}} = \frac{x_{j,\mathrm{MIN}}}{d_{j,\mathrm{MAX}}} = \frac{x[j]}{d[j] + r^{-b[j]} - ulp}, \tag{17}$$

$$\left(\frac{x}{d}\right)_{j,\mathrm{MAX}} = \frac{x_{j,\mathrm{MAX}}}{d_{j,\mathrm{MIN}}} = \frac{x[j] + r^{-a[j]} - ulp}{d[j]}. \tag{18}$$

In this case, $q_{c[j]+1} = k$ if the following is satisfied:

$$q[j] + k \cdot r^{-(c[j]+1)} \le \left(\frac{x}{d}\right)_{j,\mathrm{MIN}}, \tag{19}$$

$$\left(\frac{x}{d}\right)_{j,\mathrm{MAX}} < q[j] + (k+1) \cdot r^{-(c[j]+1)}. \tag{20}$$

Fig. 3 visualizes the condition for $q_{c[j]+1} = k$. We assume that we have found $q[j] = q_0.q_1 \cdots q_{c[j]}$ from x and d above, so the following is satisfied:

$$q[j] \le \frac{x}{d} < q[j] + r^{-c[j]}, \tag{21}$$

which is shown in Fig. 3(a). If we split the range into r sub-ranges of length $r^{-(c[j]+1)}$ and the range of $\frac{x}{d}$ is $[q[j] + k \cdot r^{-(c[j]+1)}, q[j] + (k+1) \cdot r^{-(c[j]+1)})$, then $q_{c[j]+1} = k$ as shown in Fig. 3(b). (19) and (20) show the conditions.

However, the range of $\frac{x}{d}$ might not fall into a sub-range as shown in Fig. 3(c). In this case, we cannot find $q_{c[j]+1}$ from the given x and d, and need more digits of x and/or d to find $q_{c[j]+1}$.

Now, we rewrite (19) as follows:

$$0 \le x[j] - (q[j] + k \cdot r^{-(c[j]+1)}) \cdot (d[j] + r^{-b[j]} - ulp). \tag{22}$$

Similarly, we rewrite (20) as follows:

$$x[j] + r^{-a[j]} - ulp - d[j] \cdot (q[j] + (k+1) \cdot r^{-(c[j]+1)}) < 0. \tag{23}$$

Notice that most of the terms are shifted versions of $q[j]$ and $d[j]$ except $d[j] \cdot q[j]$, which requires a multiplication.

B. Incremental Update of $d[j] \cdot q[j]$

Evaluation of (22) and (23) requires $d[j] \cdot q[j]$. However, the multiplication of $d[j]$ and $q[j]$ is too costly and cannot be completed in a cycle, so we present how to incrementally update $d[j] \cdot q[j]$ in this section.

First of all, define $m[i, j]$ as follows:

$$m[i, j] = (d_0.d_1 \cdots d_i) \cdot (q_0.q_1 \cdots q_j). \tag{24}$$

Then, we can obtain $m[i + w, j + t]$ as follows:

$$\begin{aligned} m[i+w, j+t] = m[i, j] &+ (d_0.d_1 \cdots d_i) \cdot (q_{j+1} \cdots q_{j+t}) \\ &+ r^{-(i+w)} \cdot (d_{i+1} \cdots d_{i+w}) \cdot (q_0.q_1 \cdots q_{j+t}) \end{aligned} \tag{25}$$

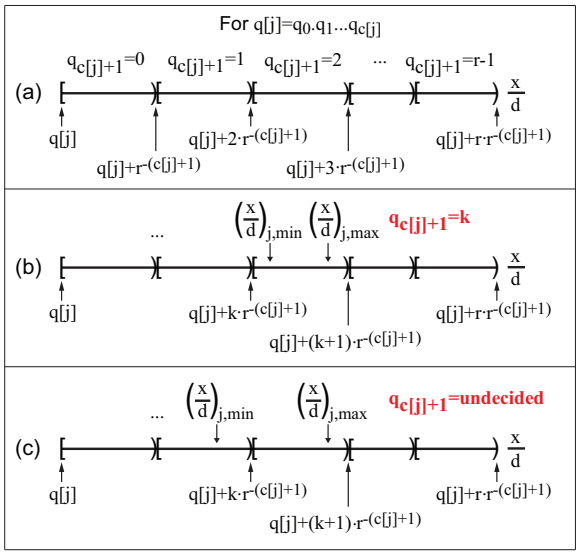

Fig. 3. Interval-analysis-based online division.

As long as w and t are small, we can incrementally compute $m[i + w, j + t]$ using $m[i, j]$, carry-save adders (CSAs), and a carry-propagate adder (CPA).

In addition, we need to incrementally update m in the two cases as follows. First, if additional digits of d are given, we should update m. Second, if we obtain an additional digit of q, we should update m. Notice that we try to obtain one digit of q in a cycle, so $t = 1$. For w, we find that $w \cdot \log_2 r \le 4$ is a proper condition for the incremental update. For example, $w \le 4$ if $r = 2$, $w \le 2$ if $r = 4$, and $w = 1$ if $r = 16$.

C. Example: Radix-4 Online Division

Suppose $n = 4$, $r = 4$, $x = 1.2002_4$, and $d = 1.0112_4$. We also assume that one digits of x and d are given every clock cycle starting from the most significant digits.

Since x and d have hidden 1's in the IEEE floating-point standard format, $x_0 = 1$ and $d_0 = 1$ are given at cycle 0 in Table II. At cycle 1, $x_1 = 2$ and $d_1 = 0$ are given. Since $x[1] = 1.2_4 > d[1] = 1.0_4$, $q[1] = q_0 = 1$.

At cycle 2, $x_2 = 0$ and $d_2 = 1$ are given. (22) and (23) for $k = 0$ are true and false, respectively. Similarly, applying them for $k = 1, 2, 3$ leads to (true, true), (false, true), and (false, true), respectively. Thus, $k = 1$ satisfies both (22) and (23), whereas all the others do not satisfy at least one of (22) and (23). Thus, $q_1 = 1$ and $q[2] = q_0.q_1 = 1.1_4$.

At cycle 3, $x_3 = 0$ and $d_3 = 1$ are given. Applying (22) and (23) for $k = 0, 1, 2, 3$ leads to (true, false), (true, false), (true, true), and (false, true), respectively. Since $k = 2$ satisfies both (22) and (23), $q_2 = 2$ ($q[3] = 1.12_4$). At cycle 4, $x_4 = 2$ and $d_4 = 2$ are given, and $k = 0$ satisfies both (22) and (23), so $q_3 = 0$ ($q[4] = 1.120_4$). Since all digits of x and d have been given, we just apply (22) and (23) for $k = 0, 1, 2, 3$ at cycle 5 and find $q_4 = 3$ ($q[5] = 1.1203_4$).

TABLE II
AN EXAMPLE OF THE ONLINE DIVISION ALGORITHM. $n = 4$, $r = 4$, $x = 1.2002_4$, AND $d = 1.0112_4$. WE ASSUME THAT ONE DIGITS OF x AND d ARE GIVEN EVERY CLOCK CYCLE.

Cycle	x	d	q
0	1_4	1_4	
1	1.2_4	1.0_4	$q = 1_4$
2	1.20_4	1.01_4	$q = 1_4$
	Apply (22) and (23). $k = 0$: (true, false), $k = 1$: (true, true), $k = 2$: (false, true), $k = 3$: (false, true) $\Rightarrow q = 1.1_4$		
3	1.200_4	1.011_4	$q = 1.1_4$
	Apply (22) and (23). $k = 0$: (true, false), $k = 1$: (true, false), $k = 2$: (true, true), $k = 3$: (false, true) $\Rightarrow q = 1.12_4$		
4	1.2002_4	1.0112_4	$q = 1.12_4$
	Apply (22) and (23). $k = 0$: (true, true), $k = 1$: (false, true), $k = 2$: (false, true), $k = 3$: (false, true) $\Rightarrow q = 1.120_4$		
5	1.2002_4	1.0112_4	$q = 1.220_4$
	Apply (22) and (23). $k = 0$: (true, false), $k = 1$: (true, false), $k = 2$: (true, false), $k = 3$: (true, true) $\Rightarrow q = 1.1203_4$		

D. Online Division with Offline Operands

Suppose x and d are fully given at cycle 0 (offline division). In this case, $a[j] = b[j] = n$ for all j, so we can rewrite (22) as follows:

$$0 \le x - (q[j] + k \cdot r^{-(c[j]+1)}) \cdot d \Leftrightarrow k \cdot d \le r^{c[j]+1} \cdot (x - d \cdot q[j]), \quad (26)$$

which is equivalent to the left side of (8) ($c[j] = j$ in this case). Similarly, we can rewrite (23) as follows:

$$r^{c[j]+1} \cdot (x - d \cdot q[j]) < (k+1) \cdot d, \quad (27)$$

which is equivalent to the right side of (8). This proves that (22) and (23) are equivalent to (8) if x and d are offline operands. Thus, the online algorithm works as an offline algorithm if x and d are fully given at cycle 0.

V. DIVIDER ARCHITECTURE AND HARDWARE IMPLEMENTATION

In this section, we present hardware implementation of the offline and online division algorithms.

A. Design Parameters, Input, and Output

1) Input: For offline division, the input consists of a dividend x and a divisor d in IEEE floating-point standard format. For online division, the input requires additional information for the validness of the digits of the dividend and divisor. If the significand of the dividend (or divisor) is $y_1 \cdots y_n$, then we assume that each digit has a separate valid bit (0: invalid, 1: valid). The execution unit generating the dividend (or divisor) has to generate the valid bits too. As shown in Table III, x and d are the dividend and the divisor, respectively, and v_x and v_d are the valid bits for x and d, respectively. For $n = 4$ and $r = 4$, for example, if $x_1 \cdots x_4 = 1200_4$ and only the first two digits are valid, then v_x is 1100.

2) Output: For offline division, the output consists of the quotient q in IEEE floating-point standard format. For online division, we also generate valid bits v_q for the quotient. Notice that the update of v_q is sometimes delayed until the last moment if rounding and normalization invert many bits. For example, suppose $n = 4$, $r = 4$, and $q = 1.03333_4$.

TABLE III
DESIGN PARAMETERS AND CONSTANTS.

	Description
x, d, q	Dividend, divisor, quotient
v_x, v_d, v_q	Valid bits for x, d, q for online division
r	Radix-r division (4 or 16)
w	# digits of d used to update m (2 or 4 bits) in (25)

Rounding of it leads to $q = 1.1000_4$, so we cannot determine the valid bits of all the four fractional digits of q before rounding. However, this does not occur often because the carry propagation due to rounding and normalization stops at digit q_i if $q_i < r - 1$. Thus, if $q[j] = q_0.q_1 \cdots q_{c[j]}$ and i is the index of the rightmost digit whose value is less than $r - 1$, then $v_{q,1} = \cdots = v_{q,i} = 1$ and $v_{q,i+1} = \cdots = v_{q,n} = 0$.

3) Design Parameters: Table III also shows two design parameters for the hardware implementation of the divider. In radix-r division, we try to obtain $\log_2 r$ bits of the quotient in a cycle. In this paper, we use 4 and 16 for r.

w is the number of unused digits of d to include in the calculation of $m = d[j] \cdot q[j]$ in (25). As explained in Section IV-B, w should be sufficiently small, so we use 2 bits or 4 bits for w.

B. Divider Architecture

Algorithm 1 shows a pseudo code of the divider. The first step (Step 1) finds q_0 by (22) and (23). First, we initialize m (Line 2) and evaluate (22) and (23) for $k = 0, \cdots, r-1$. If a certain k satisfies both, then $q_0 = k$. We also compute $m = m[b, c] = (d_0.d_1 \cdots d_b) \cdot q_0$, which requires a few carry-save adders and a carry-propagate adder. b is the index of the rightmost valid digit of the divisor d and we find it by the $idx(v_d)$ function (Line 5). We also store v_d in $v_{d,\mathrm{OLD}}$, then move on to Step 2 from the next cycle. If we cannot find q_0, we stay in Step 1 (Line 8). Regarding $m = q_0 \cdot d_{\mathrm{MIN}}$, suppose d is valid only down to the b-th digit, $d_0.d_1 \cdots d_b$. In this case, we do not assume that $d_{b+1} \cdots d_n$ is $0 \cdots 0$. Thus, when we compute m, we reset the invalid digits $d_{b+1} \cdots d_n$ of d and then multiply it by q_0. d_{MIN} is actually $d_0.d_1 \cdots d_b 0 \cdots 0$.

Step 2 finds the rest of q (Line 11 to 20). First, we compare the new v_d received in the current cycle and the previous value of v_d ($v_{d,\mathrm{OLD}}$) (Line 11). If the number of new valid digits in d received in the current cycle is greater than or equal to w in Table III, then we update m incrementally using (25) (Line 13). Also, we store the new valid bits in $v_{d,\mathrm{OLD}}$ (Line 14). For example, suppose $v_{d,\mathrm{OLD}} = 1100\ 0000$, $v_d = 1111\ 1100$, and $w = 2$. This means that $d_0.d_1d_2$ was valid previously and $d_0.d_1 \cdots d_6$ is valid in the current cycle. Since $w = 2$, we update m by $m+ = (0.00d_3d_4) \cdot q$ and $v_{d,\mathrm{OLD}}$ becomes $1111\ 0000$.

Then, we evaluate (22) and (23) for $k = 0, \cdots, r - 1$ (Line 16). If both (22) and (23) are true for a certain k, then $q_{c+1} = k$ (Line 18) and we update m incrementally by $m \leftarrow m + d_{\mathrm{MIN}} \cdot (k \cdot r^{-(c+1)})$ (Line 19). Notice that it is also possible that none of the values of k satisfies both (22) and

Input: x (dividend), d (divisor), v_x, v_d (valid bits)
Output: q, v_q
1: **// Step 1** (q_0)
2: $b \leftarrow -1, c \leftarrow -1$ for $m[b, c]$. $m \leftarrow 0$.
3: Evaluate (22) and (23) for $k = 0, \cdots r-1$.
4: **if** Both (22) and (23) are true for a certain k **then**
5: $q_0 \leftarrow k$, $m \leftarrow q_0 \cdot d_{\rm MIN}$, $c \leftarrow 0$, $b \leftarrow idx(v_d)$, $v_{d,\rm OLD} \leftarrow v_d$.
6: Execute Step 2 from the next cycle.
7: **else**
8: Stay in Step 1.
9: **end if**
10: **// Step 2** ($q_1 \cdots q_n$)
11: $new_d \leftarrow compare(v_d, v_{d,\rm OLD})$.
12: **if** $new_d \geq w$ **then**
13: Update m incrementally. $b \leftarrow b + 1$.
14: $v_{d,\rm OLD} \leftarrow arithmetic\_shift\_right(v_{d,\rm OLD}, w \text{ bits})$
15: **end if**
16: Evaluate (22) and (23) for $k = 0, \cdots, r-1$.
17: **if** Both (22) and (23) are true for a certain k **then**
18: $q_{c[j]+1} \leftarrow k$
19: Update m incrementally. $c \leftarrow c + 1$.
20: **end if**

Algorithm 1: Pseudo code of the divider architecture

(23). This means that we cannot find q_{c+1} in this cycle from the given x and d and we need more digits of x and/or d to find q_{c+1}.

C. Hardware Implementation

Fig. 4 shows the hardware implementation of the second step (Step 2) of the proposed divider. The "Comp" unit compares the current (v_d) and previous ($v_{d,\rm OLD}$) valid bits of the divisor d. If there are $\log_r w$ unused digits of $d[j]$ for the computation of m, we extract them ($d_{\rm new}$ in the figure) from $d[j]$ and update m incrementally by $m+ = q[j] \cdot d_{\rm new}$. We also generate the terms in (22) and (23) to evaluate the inequalities. Then, we add the terms by CSAs and two CPAs for each $k = 0, \cdots, r-1$. If a certain k satisfies both (22) and (23), then the next digit of q is k, so we update q. We also generate $m_k = m + (r^- c[j] + 1 \cdot k) \cdot d[j]$ by CSAs and CPAs for each k to reduce the execution time. If we find $q_{c[j]+1} = k$, we update m just by selecting m_k.

VI. Simulation Results

In this section, we present simulation results for floating-point division based on the IEEE Binary64 (double-precision) standard. We implemented all the dividers using Verilog and synthesized them using Synopsys Design Compiler and a 22nm standard cell library.

A. Dividers Used for the Comparison

We compare five offline and five online dividers shown in Table IV. AN16 is a radix-8 divider using the radix-8 digit-recurrence algorithm with a look-up table for quotient digit selection [5]. SA17 is a radix-16 divider based on a radix-16 digit-recurrence algorithm with a wide digit set [6]. JB20 is a radix-64 divider using three radix-4 iterations to obtain six quotient bits in a cycle [7]. FF2 and FF4 are the radix-4 and radix-16 normal-binary offline dividers, respectively.

AT03 is a radix-4 online divider [16]. It unfolds radix-2 recurrence equations to obtain two quotient bits in a cycle.

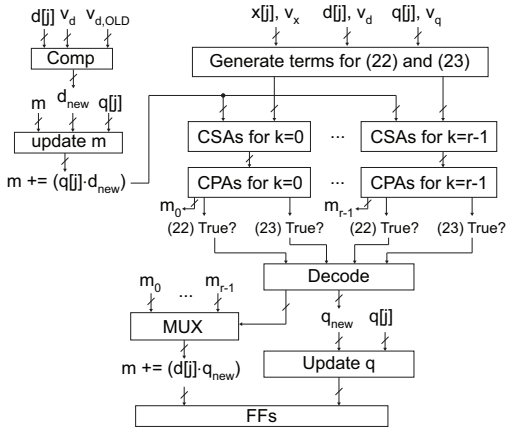

Fig. 4. Hardware implementation of the normal binary floating-point online divider.

FNyw are the proposed dividers. y is 2 (radix-4) or 4 (radix-16). w is the number of digits of d used to update m in (25). w is 2 or 4 bits.

B. Offline Division

We first compare the dividers for an offline division. Table IV shows the clock periods, execution times (of a single offline division), energy, and areas of the dividers. The numbers in the parentheses are the ratios compared to the values of the FF2 design.

1) Execution Time: FF4 has the shortest execution time for an offline division. AN16, SA17, and JB20 are 43%, 89%, and 63% slower than FF4, respectively. FF2 also shows shorter execution time than AN16, SA17, and JB20 by 10%, 44%, and 24%, respectively. FF2 has a shorter critical path than FF4, so the clock period of FF2 is 180ps shorter than that of FF4. However, FF2 finds two quotient bits per cycle while FF4 finds four quotient bits per cycle. As a result, FF4 outperforms FF2 by 24%. Notice that the cost of FF4 is more silicon area and energy consumption.

Although FF4 finds two more bits per cycle than FF2, its clock period is only 180ps longer than FF2 for the following reason. The most time-consuming part in the normal-binary offline division algorithm is the calculation of (10) and the decoding of the result. FF4 decodes a 15-bit string, whereas FF2 decodes a 3-bit string, so the 15-bit decoder adds a 180ps additional delay to the critical path. As a result, FF4 has a slightly longer clock period than FF2.

The four online dividers have longer clock periods than FF4 because they have much more complex logic and carry-save addition stages for the computation of (22) and (23). In addition, FN22 and FN24 obtain two quotient bits per cycle, so they have longer execution times than FF4 and all the other designs. However, FN42 and FF44 show execution times comparable with JB20. AT03 is 108% slower than FF4 due to its complex CSA stages.

The following shows the critical path of FF4:

$$\text{CPA} \rightarrow \text{Decoder} \rightarrow 15:1 \text{ Mux}$$

TABLE IV
A COMPARISON OF THE DIVIDERS FOR A BINARY64 OFFLINE DIVISION. THE NUMBERS IN THE PARENTHESES ARE THE RATIOS COMPARED TO THE VALUES OF THE FF2 DESIGN. BOTH THE EXECUTION TIME AND ENERGY ARE FOR ONE DIVISION OPERATION.

	Offline dividers					Online dividers				
	AN16 [5]	SA17 [6]	JB20 [7]	FF2	FF4	AT03 [16]	FN22	FN24	FN42	FN44
Radix	8	16	64	4	16	4	4	4	16	16
Clock period (ns)	0.71	1.21	1.36	0.46	0.64	0.63	0.72	0.78	0.92	0.99
Execution time (ns)	15.62 (1.10)	20.57 (1.44)	17.68 (1.24)	14.26 (1.00)	10.88 (**0.76**)	23.31 (1.63)	23.04 (1.62)	24.96 (1.75)	16.56 (1.16)	17.82 (1.25)
Energy (pJ)	49.7 (1.05)	106.3 (2.26)	62.9 (1.34)	47.1 (**1.00**)	82.1 (1.74)	163.9 (3.48)	218.0 (4.63)	319.4 (6.78)	475.8 (10.10)	569.9 (12.10)
Area (um^2)	3,871 (1.69)	12,663 (5.52)	6,489 (2.83)	2,292 (**1.00**)	9,082 (3.96)	5,217 (2.28)	11,554 (5.04)	16,705 (7.29)	50,703 (22.12)	57,974 (25.29)

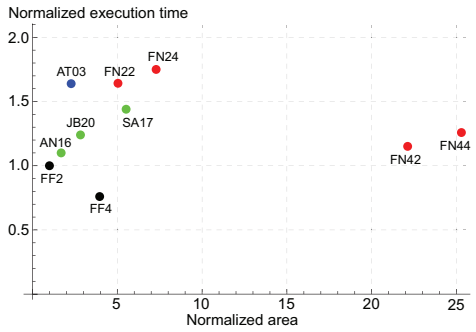

Fig. 5. Area vs. execution time for an offline division (Table IV). Both the areas and execution times are normalized to the FF2 design.

The following shows the critical path of FN44:

$$\text{Shifter} \rightarrow 5 \text{ CSAs} \rightarrow \text{CPA} \rightarrow 16:1 \text{ Mux}$$

Fig. 5 shows normalized area vs. normalized execution time of the dividers. Both the area and execution time values are normalized to the FF2 design.

2) Energy Consumption and Area: FF2 consumes the least amount of energy among all the dividers. In fact, AN16 consumes the lowest power. However, the execution time of AN16 is 15.62ns, whereas that of FF2 is 14.26ns, so AN16 consumes 9% more energy than FF2. On the other hand, the fastest offline divider FF4 consumes 74% more energy than FF2 because FF4 evaluates 15 inequalities, whereas FF2 evaluates only three inequalities for (10). For the same reason, FF4 consumes almost $4\times$ more silicon area than FF2.

The online dividers consume much more energy and silicon area than the offline dividers because they need more complex logic to handle partially-given operands. AT03 consumes $3.3\times$ and $1.9\times$ more energy than FF2 and FF4, respectively, and $2.28\times$ more silicon area than FF2. Similarly, FN22, FN24, FN42, and FN44 consume $4.6\times$ to $12.1\times$ more energy consumption and $5.0\times$ to $25.3\times$ more silicon area than FF2. However, notice that a very simple logic can be added to the FN dividers to turn off the online logic part when the operands are offline. In this case, the energy consumption will go down dramatically.

C. Online Division

We compare execution times for online division in this section. Notice that the offline dividers can start execution only when all the digits of the operands are given. Thus, their execution times are independent of the values of the operands. AT03 has a fixed execution time in a different context. It needs a so-called *online delay* (three cycles in [16]) before generating quotient digits if it receives one digits of x and d every cycle. On the contrary, the online dividers proposed in this paper have variable-length execution times because they might not be able to obtain a quotient digit in a cycle by the evaluation of (22) and (23).

We simulated the following two cases. First, two bits of x and d are given every cycle. Second, four bits of x and d are given every cycle. We also assume that the generators of x and d have the same clock period as the target divider.

Table V shows the results of the online division for the two cases. When two bits of x and d are given every cycle, AT03 receives enough digits to find quotient digits. In addition, it has a sufficiently low clock period, so it shows the shortest execution time (14% as fast as FF2). FN22 and FN24 are 4.8% and 10% slower than AT03, respectively, but they are faster than FF2 and FF4. FN22 and FN24 have shorter execution times than FN42 and FN44 because the latter do not receive enough digits per cycle to obtain quotient digits.

The trend changes dramatically if four bits of x and d are given every cycle. In this case, the execution times of FN42 and FN44 decrease significantly because they receive enough digits to find quotient digits almost every cycle. The execution times of the offline dividers also go down just because their waiting time goes down. On the other hand, AT03, FN22, and FN24 do not benefit from that because receiving two bits of x and d every cycle is already enough for them, so receiving four bits of x and d does not help them reduce the execution time further. In this simulation, FN42 and FN44 are the best designs with respect to the execution time. FN42 and FN44 are 23% and 19% as fast as AT03, respectively.

Fig. 6 shows normalized area vs. normalized execution time of the dividers for the online division case in which four bits of x and d are given every cycle. Both the area and execution time values are normalized to the FF2 design.

TABLE V
A COMPARISON OF THE DIVIDERS FOR BINARY64 ONINE DIVISION. THE NUMBERS IN THE PARENTHESES ARE THE RATIOS COMPARED TO THE VALUES OF THE FF2 DESIGN.

	Offline dividers					Online dividers				
	AN16 [5]	SA17 [6]	JB20 [7]	FF2	FF4	AT03 [16]	FN22	FN24	FN42	FN44
Radix	8	16	64	4	16	4	4	4	16	16
Execution time (ns)	34.08	52.03	53.04	26.22	27.52	23.31	23.76	24.96	29.44	30.69
Two bits of x and d are given per cycle	(1.30)	(1.98)	(2.02)	(1.00)	(1.05)	(**0.89**)	(0.91)	(0.95)	(1.12)	(1.17)
Execution time (ns)	24.85	36.30	35.36	20.24	19.20	23.31	23.76	24.96	17.48	18.32
Four bits of x and d are given per cycle	(1.23)	(1.79)	(1.75)	(1.00)	(0.95)	(1.15)	(1.17)	(1.23)	(**0.86**)	(0.90)

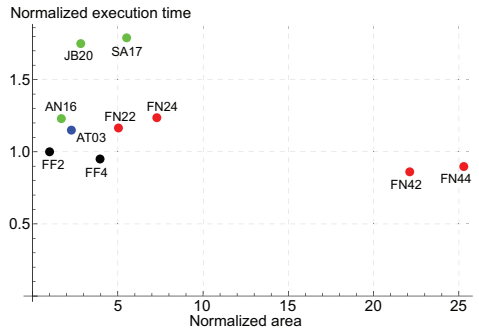

Fig. 6. Area vs. execution time of the second online division case in Table V in which four bits of x and d are given per cycle. Both the areas and execution times are normalized to the FF2 design.

D. Discussion

The execution times of the online dividers are dependent on the speed of the generators (the units sending x and d to the online dividers). When an online divider computes a quotient digit, it would not be able to find the digit if a sufficient number of digits of the operands are not given to the divider. Thus, if the generators are low-performance units, then the execution times of the online dividers would go up in Table V because the dividers sometimes have to wait for more operand digits. On the contrary, even if the performance of the generators goes up, the execution times of the online dividers would not go up infinitely. The best cases for the online dividers would be the offline cases shown in Table IV.

If the operands are irregularly given, the online dividers might be able to benefit from that significantly. For example, suppose most of the digits of the operands have been given, but only a few last digits are missing. In this case, the offline dividers still have to wait for the last digits, but the online dividers can find most of the quotient digits. When the missing digits are given, the online dividers can finish the division in just a few cycles.

VII. CONCLUSION

In this paper, we proposed a normal-binary floating-point dual-purpose division algorithm for both offline and online division. The algorithm uses interval analysis to find quotient digits from offline or online operands. The algorithm uses the conventional binary number system and does not require any convergence check. If four bits of the operands are given every cycle, FN42 and FN44 achieve the shortest execution time at the cost of larger area and more energy consumption (due to the concurrent evaluation of the inequalities).

ACKNOWLEDGMENT

This work was supported by the Defense Advanced Research Projects Agency (DARPA) Young Faculty Award under Grant D16AP00119.

REFERENCES

[1] S. F. Oberman and M. J. Flynn, "Design Issues in Division and Other Floating-Point Operations," in *IEEE Trans. on Computers*, vol. 46, no. 2, Feb. 1997, pp. 154–161.
[2] E. Matthews, A. Lu, Z. Fang, and L. Shannon, "Rethinking Integer Divider Design for FPGA-based Soft-Processors," in *Proc. IEEE Annual Int. Symp. on Field-Programmable Custom Computing Machines*, 2019, pp. 289–297.
[3] U. S. Patankar and A. Koel, "Review of Basic Classes of Dividers Based on Division Algorithm," in *IEEE Access*, vol. 9, Jan. 2021, pp. 23 035–23 069.
[4] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean *et al.*, "The Microarchitecture of the Pentium 4 Processor," in *Intel Technology Journal Q1*, 2001, pp. 1–13.
[5] A. Nannarelli, "Performance/Power Space Exploration for Binary64 Division Units," in *IEEE Trans. on Computers*, vol. 65, no. 5, May 2016, pp. 1671–1677.
[6] S. Amanollahi and G. Jaberipur, "Energy-Efficient VLSI Realization of Binary64 Division with Redundant Number Systems," in *IEEE Trans. on VLSI Systems*, vol. 25, no. 3, Mar. 2017, pp. 954–961.
[7] J. D. Bruguera, "Low Latency Floating-Point Division and Square Root Unit," in *IEEE Trans. on Computers*, vol. 69, no. 2, Feb. 2020, pp. 274–287.
[8] F. Lyu, Y. Xia, Y. Chen, Y. Wang, Y. Luo *et al.*, "High-Throughput Low-Latency Pipelined Divider for Single-Precision Floating-Point Numbers," in *IEEE Trans. on VLSI Systems*, vol. 30, no. 4, Apr. 2022, pp. 544–548.
[9] J. D. Bruguera, "Low-Latency and High-Bandwidth Pipelined Radix-64 Division and Square Root Unit," in *Proc. IEEE Int. Symp. on Computer Arithmetic*, 2022, pp. 10–17.
[10] K. Trivedi and M. Ercegovac, "On-Line Algorithms for Division and Multiplication," in *IEEE Trans. on Computers*, vol. C-26, no. 7, Jul. 1977, pp. 681–687.
[11] K. Trivedi, "Higher Radix On-Line Division," in *Proc. IEEE Int. Symp. on Computer Arithmetic*, 1978, pp. 164–174.
[12] O. Watanuki and M. Ercegovac, "Floating-Point On-Line Arithmetic: Algorithms," in *Proc. IEEE Int. Symp. on Computer Arithmetic*, 1981, pp. 81–86.
[13] O. Watanuki and M. Ercegovac, "Floating-Point On-Line Arithmetic: Error Analysis," in *Proc. IEEE Int. Symp. on Computer Arithmetic*, 1981, pp. 87–91.
[14] M. Ercegovac, "On-Line Arithmetic: An Overview," in *Real Time Signal Processing VII: Proc. SPIE*, vol. 495, 1984, pp. 86–93.
[15] P. K.-G. Tu and M. D. Ercegovac, "A Radix-4 On-Line Division Algorithm," in *Proc. IEEE Int. Symp. on Computer Arithmetic*, 1987, pp. 181–187.
[16] A. F. Tenca, A. Shantilal, and M. Sinky, "A Radix-4 On-line Division Design and Its Application to Networks of On-line Modules," in *Proceedings of SPIE*, vol. 5205, 2003, pp. 529–540.
[17] M. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kauffmann, 2004.
[18] M. Ercegovac, "An On-Line Square Root Algorithm," in *IEEE Trans. on Computers*, vol. C-31, no. 1, Jan. 1982, pp. 70–75.
[19] M. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementation*. Kluwer Academic Publishers, 1994.
[20] "IEEE Standard for Floating-Point Arithmetic," http://ieee.org/.