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

Jihee Seo\textsuperscript{1,2} and Dae Hyun Kim\textsuperscript{2}
\textsuperscript{1}Synopsys, Inc., Hillsboro, OR, USA
\textsuperscript{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):

\[
\sqrt{a \over b} \cdot (c \cdot \cos x + d \cdot e \cdot \sin y).
\]

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 \( \sqrt{a/b} \).
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 the DD dependency reduces the execution time of the Bessel function benchmark by 2.96%.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MM</th>
<th>DM</th>
<th>MD</th>
<th>DD</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEM</td>
<td>0%</td>
<td>0.58%</td>
<td>3.85%</td>
<td>0.36%</td>
<td>4.79%</td>
</tr>
<tr>
<td>EM simulation</td>
<td>5.71%</td>
<td>0.25%</td>
<td>0.33%</td>
<td>0.43%</td>
<td>1.57%</td>
</tr>
<tr>
<td>Statistical inference</td>
<td>0.56%</td>
<td>0%</td>
<td>1.47%</td>
<td>2.96%</td>
<td>28.04%</td>
</tr>
<tr>
<td>Bessel function</td>
<td>23.61%</td>
<td>0%</td>
<td>1.47%</td>
<td>2.96%</td>
<td>28.04%</td>
</tr>
</tbody>
</table>

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

C. Digit-Recurrence Floating-Point Division

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

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

and a digit-selection function \(\text{SEL}(w[j], d)\) [19]. Each digit \(q_i\) is one of \(D_a = \{-a, -a+1, \ldots -a-1, a\}\) 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 $x = d \cdot q + \text{rem}$, $x$, $d$, 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:

$$0 \leq \text{rem} < d \cdot \text{ulp}.$$  \hfill (3)

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

$$q[j] = q_0 q_1 \cdots q_j = \sum_{i=0}^{j} q_i \cdot r^{-i}.$$  \hfill (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 $x = d \cdot q + \text{rem}$, the followings should be satisfied for each $j$ [19]:

$$0 \leq \text{rem}[j] < d \cdot r^{-j} \iff 0 \leq x - d \cdot q[j] < d \cdot r^{-j}$$

$$0 \leq r^{j+1} \cdot (x - d \cdot q[j + 1]) < d.$$  \hfill (5)

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

$$0 \leq r^{j+1} \cdot (x - d \cdot q[j + 1]) < d \iff 0 \leq r^{j+1} \cdot (x - d \cdot q[j]) < d \iff 0 \leq r^{j+1} \cdot (x - d \cdot q[j]) - d \cdot q_{j+1} < d.$$  \hfill (6)

Define $w[j]$ as follows:

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

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

$$0 \leq r^{j+1} \cdot (x - d \cdot q[j]) - k \cdot d < d.$$  \hfill (8)

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

$$0 \leq r \cdot w[j] - 1 \cdot d,$$
$$0 \leq r \cdot w[j] - 2 \cdot d,$$
$$\cdots$$
$$0 \leq r \cdot w[j] - (r - 1) \cdot d.$$  \hfill (10)

Suppose $f_k = 1$ (or 0) if $0 \leq r \cdot w[j] - k \cdot d$ is true (or false). Then, the bit string $F = (f_1 f_2 \cdots f_{r-1})$ is one of $(000 \cdots 0)\), $(100 \cdots 0), (110 \cdots 0), (111 \cdots 1)$. 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 $x \geq d$, $q_0 = 1$ and $w[0] = 0.1230_4$. Applying (10) leads to the following:

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

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:

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

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

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

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

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

$F = (111)$, so $q_4 = 3, q = 1.1203_4$ and $\text{rem} = x - d \cdot q = 0.00001332_4 < d \cdot \text{ulp} = 0.00012002_4$, 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 $x = d \cdot q + \text{rem}$, suppose we have obtained $q[j] = q_0 q_1 \cdots q_j$. This means that $x/d$ satisfies the following:

$$q[j] \leq \frac{x}{d} < q[j] + r^{-j}.$$  \hfill (11)

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

$$q[j] + k \cdot r^{-(j+1)} < \frac{x}{d} < q[j] + (k + 1) \cdot r^{-(j+1)}.$$  \hfill (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 $x = d \cdot q + \text{rem}$, 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:

$$x[j] = x_0 x_1 \cdots x_n[j] = \sum_{i=0}^{n[j]} x_i \cdot r^{-i},$$  \hfill (13)

$$d[j] = d_0 d_1 \cdots d_n[j] = \sum_{i=0}^{n[j]} d_i \cdot r^{-i},$$  \hfill (14)

$$q[j] = q_0 q_1 \cdots q_n[j] = \sum_{i=0}^{n[j]} q_i \cdot r^{-i}.$$  \hfill (15)
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:

\[
\left( \frac{x}{d} \right)_{j,\text{MIN}} = \frac{x[j] - r^{-c[j]+1}}{d[j]} - \ulp,
\]

\[
\left( \frac{x}{d} \right)_{j,\text{MAX}} = \frac{x[j] + r^{-n} - r^{-c[j]+1}}{d[j]} - \ulp.
\]

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

\[
q[j] + k \cdot r^{-c[j]+1} \leq \left( \frac{x}{d} \right)_{j,\text{MIN}},
\]

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

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:

\[
q[j] \leq \frac{x}{d} < q[j] + r^{-c[j]+1},
\]

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:

\[
0 \leq x[j] - (q[j] + k \cdot r^{-c[j]+1}) \cdot (d[j] + r^{-n} - \ulp).
\]

Similarly, we rewrite (20) as follows:

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

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:

\[
m[i, j] = (d_0, d_1 \cdots d_i) \cdot (q_0 q_1 \cdots q_j) \cdot d_{i+1} \cdots d_{j+1} \cdot q_{j+1} \cdots q_t.
\]

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

\[
m[i + w, j + t] = m[i, j] + (d_{i+w} \cdots d_{j+t}) \cdot (q_{j+1} \cdots q_{j+t}) + r^{-i} \cdot (d_{i+1} \cdots d_{i+w}) \cdot (q_0 \cdots q_{j+t}).
\]

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 \( w \cdot \log_2 r \leq 4 \) is a proper condition for the incremental update. For example, \( w \leq 4 \) if \( r = 2 \), \( w \leq 2 \) if \( r = 4 \), and \( w = 1 \) if \( r = 16 \).

C. Example: Radix-4 Online Division

Suppose \( n = 4 \), \( r = 4 \), \( x = 1.2002 \), and \( d = 1.0110 \). 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 \( x[1] = 1.24 > d[1] = 1.04 \), \( q[1] = q_0 = 1 \).

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 \( q[2] = q_0 q_1 = 1.1q_1 \).

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), \( q_2 = 2 \) \( q[3] = 1.12q_1 \). At cycle 4, \( x_4 = 2 \) and \( d_4 = 2 \) are given, and \( k = 0 \) satisfies both (22) and (23), so \( q_3 = 0 \) \( q[4] = 1.12q_1 \). 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 \( q_4 = 3 \) \( q[5] = 1.1203q_4 \).
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:

\[
0 \leq x - (q[j] + k \cdot r \cdot (c[j] + 1)) \cdot d \iff k \cdot d \leq p[j] + 1 \cdot (x - d \cdot q[j]),
\]

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

\[
p[j] + 1 \cdot (x - d \cdot q[j]) < (k + 1) \cdot d,
\]

(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 \( x_1 \cdots x_4 = 12001 \) and only the first two digits are valid, then \( v_x = 1100 \).

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 \( q = 1.0333\bar{3} \).

Rounding of it leads to \( q = 1.1000_3 \), 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 \( q[j] = q_0 q_1 \cdots q_{j-1} q_j \) and \( i \) is the index of the rightmost digit whose value is less than \( r - 1 \), then \( v_{q,i} = \cdots = v_{q,i} = 1 \) 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 \( id_x(v_d) \) function (Line 5). We also store \( v_d \) in \( v_{d,OLD} \), then move on to Step 2 from the next cycle. If we cannot find \( q_0 \), we stay in Step 1 (Line 8). Regarding \( m = q_0 \cdot d_{MIN} \), suppose \( d \) is valid only down to the \( b \)-th digit, \( d_b \cdot d_{1} \cdots d_b \). In this case, we do not assume that \( d_{b+1} \cdots d_0 \) is 0 \cdots 0. Thus, when we compute \( m \), we reset the invalid digits \( d_{b+1} \cdots d_0 \) of \( d \) and multiply it by \( q_0 \). \( d_{MIN} \) is actually \( d_0 d_1 \cdots d_0 \cdot 0 \cdots 0 \).

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 \( v_d \) (\( v_{d,OLD} \)) (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 \( v_{d,OLD} \) (Line 14). For example, suppose \( v_{d,OLD} = 1100 0000 \), \( v_d = 1111 1100 \), and \( w = 2 \). This means that \( d_0 d_1 d_2 \) was valid previously and \( d_0 d_1 \cdots d_6 \) is valid in the current cycle. Since \( w = 2 \), we update \( m \) by \( m = (0.00d_3 d_4) \cdot q \) and \( v_{d,OLD} \) 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 \( m \leftarrow m + d_{MIN} \cdot (k \cdot r^{-c-1}) \) (Line 19). Notice that it is also possible that none of the values of \( k \) satisfies both (22) and
(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 ($v_d,OLD$) valid bits of the divisor $d$. If there are $\log_w w$ unused digits of $d[j]$ for the computation of $m$, we extract them ($d_{new}$ in the figure) from $d[j]$ and update $m$ incrementally by $m += q[j] \cdot d_{new}$. 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 $k = 0, \ldots, r - 1$. 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.

FNyw is 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 FN44 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.

<table>
<thead>
<tr>
<th>Offline dividers</th>
<th>Online dividers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radix</td>
<td>8</td>
</tr>
<tr>
<td>Clock period (ns)</td>
<td>0.71</td>
</tr>
<tr>
<td>Execution time (ns)</td>
<td>15.62</td>
</tr>
<tr>
<td>Energy (pJ)</td>
<td>147.1</td>
</tr>
</tbody>
</table>

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

The following shows the critical path of FN44:

Shifter → 5 CSAs → CPA → 16 : 1 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 × 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 × 1.9 × more energy than FF2 and FF4, respectively, and 2.28 × more silicon area than FF2. Similarly, FN22, FN24, FN42, and FN44 consume 4.6 × to 12.1 × more energy consumption and 5.0 × to 25.3 × 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 digit 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 FN42 are 4.8% and 10% slower than AT03, respectively, but they are faster than FF2 and FF4. FN22 and FN42 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 FN42 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 online division. The numbers in the parentheses are the ratios compared to the values of the FF2 design.

<table>
<thead>
<tr>
<th>Radix</th>
<th>Offline dividers</th>
<th>Online dividers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>Execution time (ns) Two bits of x and d are given per cycle</td>
<td>34.08</td>
<td>52.03</td>
</tr>
<tr>
<td>Execution time (ns) Four bits of x and d are given per cycle</td>
<td>24.85</td>
<td>36.30</td>
</tr>
</tbody>
</table>

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 divisor. 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