Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2013, Article ID 453173, 16 pages
http://dx.doi.org/10.1155/2013/453173

Research Article

Analysis of Fast Radix-10 Digit Recurrence Algorithms for Fixed-Point and Floating-Point Dividers on FPGAs

Malte Baesler and Sven-Ole Voigt

Institute for Reliable Computing, Hamburg University of Technology, Schwarzenbergstraße 95, 21073 Hamburg, Germany

Correspondence should be addressed to Malte Baesler; malte.baesler@tu-harburg.de

Received 3 May 2012; Accepted 17 September 2012

Academic Editor: René Cumplido

Decimal floating point operations are important for applications that cannot tolerate errors from conversions between binary and decimal formats, for instance, commercial, financial, and insurance applications. In this paper we present five different radix-10 digit recurrence dividers for FPGA architectures. The first one implements a simple restoring shift-and-subtract algorithm, whereas each of the other four implementations performs a nonrestoring digit recurrence algorithm with signed-digit redundant quotient calculation and carry-save representation of the residuals. More precisely, the quotient digit selection function of the second divider is implemented fully by means of a ROM, the quotient digit selection function of the third and fourth dividers are based on carry-propagate adders, and the fifth divider decomposes each digit into three components and requires neither a ROM nor a multiplexer. Furthermore, the fixed-point divider is extended to support IEEE 754-2008 compliant decimal floating-point division for decimal64 data format. Finally, the algorithms have been synthesized on a Xilinx Virtex-5 FPGA, and implementation results are given.

1. Introduction

Many applications, particularly commercial and financial applications, require decimal floating-point operations to avoid errors from conversions between binary and decimal formats. This paper presents five different decimal fixed-point dividers and analyzes their performances and resource requirements on FPGA platforms. All five architectures apply a radix-10 digit recurrence algorithm but differ in the quotient digit selection (QDS) function.

The first fixed-point divider (type1) implements a simple shift-and-subtract algorithm. It is characterized by an unsigned and nonredundant quotient digit calculation. Nine divisor multiples are precomputed, and in each iteration step nine carry-propagate subtractions are performed on the residual. Finally, the smallest, nonnegative difference is selected by a large fan-in multiplexer. This type1 implementation is characterized by a high area use.

The second divider (type2) uses a signed-digit quotient calculation with a redundancy of $\rho = 8/9$ and operands scaling to get a normalized divisor in the range of $0.4 \leq \text{divisor} < 1.0$. The quotient digit selection (QDS) function can be implemented fully by a ROM because it depends only on the two most significant digits (MSDs) of the residual as well as the divisor. The residual uses a redundant carry-save representation but, because of performance issues, the two MSDs are implemented by a nonredundant radix-2 representation.

The quotient digit selection (QDS) functions of the third and fourth divider (type3.a and type3.b) are based on comparators for the two most significand digits. The comparators consist of short binary carry-propagate adders (CPA), which can be implemented very efficiently in the FPGA's slice structure. The corresponding comparative values depend on the divisor's value and are precomputed and stored in a small ROM. The redundancy is $\rho = 8/9$; thus, 17 binary CPAs are required. Similar to the type2 divider, the type3.a and type3.b dividers use prescaling of the divisor $0.4 \leq \text{divisor} < 1.0$, a redundant carry-save representation of the residual, and a nonredundant radix-2 representation of the two MSDs. The dividers type3.a and type3.b differ only in the implementation of large fan-in multiplexers in the digit recurrence step. The type3.a divider implements multiplexers that minimize the LUT usage but have long latencies, whereas

the type3.b divider implements a faster dedicated multiplexer that exploits the FPGA's internal carry chains.

The quotient digit selection function of the last divider (type4) requires neither a ROM nor a multiplexer. It is characterized by divisor scaling ($0.4 \leq \text{divisor} < 0.8$) and a signed-digit redundant quotient calculation with a redundancy of $\rho = 8/9$. The quotient digit is decomposed into three components having values $\{-5, 0, 5\}$, $\{-2, 0, 2\}$, and $\{-1, 0, 1\}$. The components are computed one by one, whereby the digit selection function is constant; that is, the selection constants do not depend on the divisor's value. Similar to the type2, type3.a, and type3.b dividers, the type4 divider uses a carry-save representation for the residual with a nonredundant radix-2 representation of the two MSDs.

The fixed-point division algorithms are implemented and analyzed on a Virtex-5 FPGA. Finally, the type2 divider, which shows the best tradeoff in area and delay, is extended to a floating-point divider that is fully IEEE 754-2008 compliant for decimal64 data format, including gradual underflow handling and all required rounding modes.

The architectures of the type1, type2, and type4 dividers have already been published in [1]. However, this paper gives a more detailed description of the previous research and introduces two new dividers, which fill the design gap between the type1 and type2 dividers because they are based on two extreme examples of algorithms: the type1 divider implements a restoring quotient digit selection (QDS) function that requires nine decimal carry-propagate adders (CPAs) of full precision, whereas the nonrestoring QDS function of the type2 divider is implemented fully by means of a ROM with limited precision. In comparison, the new dividers implement nonrestoring QDS functions that are based on fast *binary* CPAs with *limited precision*.

The outline of this paper is given as follows: Section 2 motivates the use of decimal floating-point arithmetic and its advantage compared to binary floating-point arithmetic. The underlying decimal floating-point standard IEEE 754-2008 is introduced in Section 3. The digit recurrence division algorithms as well as the five different architectures of fixed-point dividers are presented in Section 4. These fixed-point dividers are extended to a decimal floating-point divider in Section 5. Postplace and route results are presented in Section 6, and finally in Section 7 the main contributions of this paper are summarized.

2. Decimal Arithmetic

Since its approval in 1985, the *binary* floating-point standard IEEE 754-1985 [2] is the most widely used implementation of floating-point arithmetic and the dominant floating-point standard for all computers. In contrast to binary arithmetic, decimal units are more complex, require more area, and are more expensive, and the simple binary coded decimal (BCD) data format has a storage overhead of approximately 20%. Thus, at that time of approval the use of binary in preference to decimal floating-point arithmetic was justified by the better efficiency.

Most people in the world think in decimal arithmetic. These decimal numbers must be converted to binary numbers when using a computer. However, some common finite numbers can only be *approximated* by binary floating-point numbers. The decimal number 0.1, for example, has a periodical continued fraction $0.1_{10} = 0.000\overline{1100}_2$. It cannot be represented exactly in a binary floating-point arithmetic with finite precision, and the conversion causes rounding errors.

As a consequence, binary floating-point arithmetic cannot be used for any calculations which do not tolerate conversion errors between decimal and binary numbers. These are, for instance, financial and business applications that even require decimal arithmetic by law [3]. Therefore, commercial application often use nonstandardized software to perform decimal floating-point arithmetic. However, these software implementations are usually from 100 to 1000 times slower than equivalent binary floating-point operations in hardware [3].

Because of the increasing importance, specifications for decimal floating-point arithmetic have been added to the IEEE 754-2008 standard for floating-point arithmetic [4] that has been approved in 2008 and offers a more profound specification than the former radix-independent floating-point arithmetic IEEE 854-1987 [5]. Therefore, new efficient algorithms have to be investigated, and providing hardware support for decimal arithmetic is becoming more and more a topic of interest.

IBM has responded to this market demand and integrates decimal floating-point arithmetic in recent processor architectures such as z9 [6], z10 [7], Power6 [8], and Power7 [9]. The Power6 is the first microprocessor that implements IEEE 754-2008 decimal floating-point format fully in hardware, while the earlier released z9 already supports decimal floating-point operations but implements them mainly in millicode. Nevertheless, the Power6 decimal floating-point unit is as small as possible and is optimized to low cost. It reuses registers from the binary floating-point unit, and the computing unit mainly consists of a wide decimal adder. Thus, its performance is rather low. Other floating-point operations such as multiplication and division are based on this adder and are performed sequentially. The decimal floating-point units of z10 and Power7 are designed similarly to those of the Power6 [7, 9].

3. IEEE 754-2008

The floating-point standard IEEE 754-2008 [4] has revised and merged the IEEE 754-1985 standard for binary floating-point arithmetic [2] and IEEE 854-1987 standard for radix-independent floating-point arithmetic [5]. As a consequence, the choice of radices has been focused on two formats: binary and decimal. In this paper we consider only the decimal data format. A decimal number is defined by the triple consisting of the sign (s), significand (c), and exponent (q):

$$x = (-1)^s \cdot c \cdot 10^q, \tag{1}$$

with $s \in \{0, 1\}$, $c \in [0, 10^p - 1] \cap \mathbb{N}$, and $q \in [q_{\min}, q_{\max}] \cap \mathbb{Z}$. IEEE 754-2008 uses two different designators for the exponent (q and e) as well as for the significand (c and m). The exponent e is applied when the significand is regarded as an

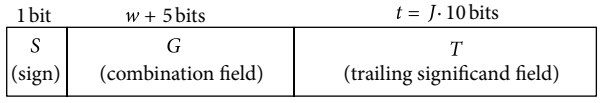

Figure 1: Decimal interchange formats [4].

Table 1: Interchange format parameters [4].

Parameter	dec32	dec64	dec128
k : storage width (bits)	32	64	128
p: precision (digits)	7	16	34
$q_{\min}$: min. exponent	−101	−398	−6176
$q_{\max}$: max. exponent	90	369	6111
Bias $= E - q$	101	398	6176
s: sign bit	1	1	1
$w + 5$: combination field (bits)	11	13	17
t: trailing significand field (bits)	20	50	110

integer digit and fraction field, denoted by m. The exponent q is applied when the significand is regarded as an integer, denoted by c. The relation is given through

$$e = q + p - 1, \qquad m = c \cdot b^{-p+1}. \tag{2}$$

In this paper we exclusively use the integer representation c with the exponent q.

Unlike binary floating-point format, the decimal floating-point number is not necessarily normalized. This leads to a redundancy, and a decimal number might have multiple representations. The set of representations is called the floating-point number's *cohort*. For example, the numbers $123 \cdot 10^0$, $1230 \cdot 10^{-1}$, and $12300 \cdot 10^{-2}$ are all members of the same cohort. More precisely, if a number has $n \leq p$ significant digits, the number of representations is $p - n + 1$.

IEEE 754-2008 defines three decimal interchange formats (*decimal32*, *decimal64*, and *decimal128*) of fixed width 32, 64, and 128 bits. As depicted in Figure 1, a floating-point number is encoded by three fields: the *sign bit* s, the *combination field* G, and the *trailing significand field*. The combination field encodes whether the number is finite, infinite, or not a number (NaN). Furthermore, in case of finite numbers the combination field comprises the biased exponent ($E = q +$ bias) and the most significant digit (MSD) of the significand. The remaining $3 \cdot J$ digits are encoded in the trailing significand field of width $t = J \cdot 10$. The trailing significand can either be implemented as a binary integer or as a densely packed decimal (DPD) number [4]. Binary encoding makes software implementations easier, whereas DPD encoding is favored by hardware implementations, as it is the case in this paper.

The encoding parameters for the three fixed-width interchange formats are summarized in Table 1. This paper focuses on the data format *decimal64* with DPD coded significand. DPD encodes three decimal digits (four bits each) into a declet (10 bits) and vice versa [4]. It results in an storage overhead of only 0.343% per digit.

IEEE 754-2008 defines five rounding modes. These are two modes to the nearest (*round ties to even* and *round ties to away*) and three directed rounding modes (*round toward positive*, *round toward negative*, and *round toward zero*) [4]. As floating-point operations are obtained by first performing the exact operation in the set of real numbers and then mapping the exact result onto a floating-point number, rounding is required whenever all significant digits cannot be placed in a single word of length p. Moreover, inexact, underflow, or overflow exceptions are signaled when necessary.

4. Decimal Fixed-Point Division

Oberman and Flynn [10] distinguish five different classes of division algorithms: digit recurrence, functional iteration, very high radix, table look-up, and variable latency, whereby many practical algorithms are combinations of multiple classes. Compared to binary arithmetic, decimal division is more complex. Currently, there are only a few publications concerning radix-10 division.

Wang and Schulte [11] describe a decimal divider based on the Newton-Raphson approximation of the reciprocal. The latency of a decimal Newton-Raphson approximation directly depends on the latency of the decimal multiplier. A pipelined multiplier has the advantage that more than one division operation can be processed in parallel; otherwise the efficiency is poor. However, the algorithm lacks remainder calculation, and the rounding is more complex. The first FPGA-based decimal Newton-Raphson dividers are presented in [12]. The dividers as well as the underlying multipliers are sequential; hence, these dividers have a high latency.

Digit recurrence division is the most widely implemented class of division algorithms. It is an iterative algorithm with linear convergence; that is, a fixed number of quotient digits is retired every iteration step. Compared to radix-2 arithmetic, radix-10 digit recurrence division is more complex because, on the one hand, decimal logic is less efficient by itself and, on the other hand, the range of the quotient digit selection (QDS) function comprises a larger digit set. Therefore, the performance of the digit recurrence divider depends on the choice of the QDS function and the implementation of decimal logic.

Nikmehr et al. [13] select quotient digits by comparing the truncated residual with limited precision multiples of the divisor. Lang and Nannarelli [14] replace the divisor's multiples by comparative values obtained by a look-up table and decompose the quotient digit into a radix-2 digit and a radix-5 digit in such a way that only five and two times the divisor are required. Vázquez et al. [15] take a different approach: the selection constants in the QDS function are obtained of truncated multiples of the divisor, avoiding look-up tables. Therefore, the multiples are computed on-the-fly. Moreover, the digit recurrence iteration implements a slow carry-propagate adder, but an estimation of the residual is computed to make the determination of the quotient digits independent of this carry-propagate adder. The decimal divider of the Power6 microprocessor [16] uses extensive prescaling to bound the divisor to be greater than or equal

to 1.0 but lower than 1.1112 in order to simplify the digit selection function. However, this prescaling is very costly, it requires a 2-digit multiply, and it needs overall six cycles on each operand. Furthermore, the digit selection function still requires a look-up table.

The first decimal fixed-point divider designed for FPGAs applies digit recurrence algorithm and is proposed by Ercegovac and McIlhenny [17, 18]. However, it has a poor cycle time, because it does not fit well on the slice structure of FPGAs but would probably fit well into ASIC designs with dedicated routing.

4.1. Digit Recurrence Division. In the following, we use different decimal number representations. A decimal number B is called BCD-$\beta_0\beta_1\beta_2\beta_3$ coded when B can be expressed by

$$B = \sum_{i=0}^{p-1} B_i \cdot 10^i$$
$$\text{with } B_i = \sum_{k=0}^{3} B_{ik} \cdot \beta_k, B_{ik} \in \{0, 1\}. \tag{3}$$

A number representation is called redundant if one or more digits have multiple representations.

Moreover, we consider a division $z = x/d$ in which the decimal dividend x and the decimal divisor d are positive and normalized fractional numbers of precision p, that is, $0.1 \le x,\ d < 1$. The quotient z and the remainder rem are calculated as follows:

$$x = z \cdot d + \text{rem}, \quad \text{rem} \le d \cdot \text{ulp}, \quad \text{ulp} = 10^{-p}. \tag{4}$$

Then the radix-10 digit recurrence is implemented by

$$w[j+1] = 10w[j] - z_{j+1} \cdot d, \quad j = 0, 1, \ldots, p \tag{5}$$

with the initial residual $w[0] = x/10$ and with the quotient digit calculated by the selection function

$$z_{j+1} = \text{SEL}(10w[j], d)$$
$$\text{with } z = z_1.z_2z_3 \cdots z_p = \sum_{j=0}^{p-1} z_{j+1} \cdot 10^{-j}. \tag{6}$$

Digit recurrence division is subdivided into the classes of restoring and nonrestoring algorithms, that differ in the quotient digit selection (QDS) function and the dynamical range of the residual. A restoring divider selects the next positive quotient digit $0 \le z_{j+1} \le 9$ such that the next partial residual is as small as possible but still positive. The IBM z900 architecture, for instance, implements such a decimal restoring divider [19]. By contrast, the QDS function of a decimal nonrestoring divider uses a digit set that is positive as well as negative $-a \le z_{j+1} \le a$, and the partial residual $w[j+1]$ might also be negative.

One advantage of restoring division is the enhanced performance since estimates of limited precision ($\widehat{w[j]} \approx w[j]$ and $\widehat{d} \approx d$) might be used in the QDS function SEL $(\widehat{10w[j]}, \widehat{d})$. This performance gain is achieved by using a redundant digit set, $-a \le z_j \le a$, which defines a redundancy factor

$$\rho = \frac{a}{10-1} = \frac{a}{9}. \tag{7}$$

Then, in each iteration step the quotient digit z_{j+1} should be selected such that the next residual $w[j+1]$ is bounded by

$$-\rho d \le w[j+1] \le \rho d, \tag{8}$$

which is called the convergence condition [20].

This paper presents five different decimal fixed-point dividers that are described in the following. The first divider (type1) is a restoring divider whereas the four others (type2, type3.a, type3.b, and type4) are nonrestoring dividers.

4.2. Type1 QDS Function. The QDS function of the type1 algorithm is very simple. Nine multiples of the divisor $(1d, \ldots, 9d)$ are subtracted in parallel from the residual by carry-propagate adders (CPAs), and the smallest positive result is selected. The CPAs exploit the FPGA's internal fast carry logic, as described in [21].

The nine multiples are precomputed and are composed of the multiples $1d$, $2d$, $5d$, $10d$ and their negatives (10's complement), which require at most one additional CPA per multiple. The multiples $2d$ and $5d$ can be easily computed by digit recoding and constant shift operations

$$(X)_{\text{BCD-5421}} \ll 1 \equiv (X \cdot 2)_{\text{BCD-8421}}, \tag{9}$$

$$(X)_{\text{BCD-8421}} \ll 3 \equiv (X \cdot 5)_{\text{BCD-5421}}, \tag{10}$$

where (9) is read as follows. A BCD-5421 coded number X left-shifted by one bit is equivalent to the corresponding BCD-8421-coded number multiplied by two. In a similar fashion we obtain a multiplication by five using (10).

The implementation of the type1 digit recurrence divider is characterized by a high area use, due to the utilization of nine parallel CPAs. The corresponding algorithm of the type1 digit recurrence is shown in Algorithm 1.

4.3. Type2 QDS Function. The approach of the type2 division algorithm is based on the implementation of the QDS function fully by a ROM. This ROM is addressed by estimates of the residual and divisor. The use of estimates is feasible because a signed digit set together with a redundancy greater than 1/2 is used. The estimates $\widehat{10w[j]}$ and $\widehat{d}$ are obtained by truncation; that is, only a limited number of MSDs of the residual $10w[j]$ and divisor d are regarded. Furthermore, the residual is implemented in BCD-4221 carry-save representation such that the maximum error introduced by an estimation with precision t (one integer digit and $t-1$ fractional digits) is bounded by $\epsilon = 2 \cdot 10^{-t+1}$. Negative residuals are represented by their 10's complement.

The divisor is subdivided into subranges $[d_i, d_{i+1})$ of equal width $d_{i+1} - d_i = 10^{-\delta}$. For each subrange i, the QDS function $z_{j+1} = \text{SEL}_{\text{ROM}}(\widehat{10w[j]}, \widehat{d})$ is defined by selection constants $m_k(i)$:

$$z_{j+1} = k \Longleftrightarrow m_k(i) \le \widehat{10w[j]} < m_{k+1}(i). \tag{11}$$

(1) Compute $\delta_1 = 10w[j] - d$
$\vdots$
$\delta_9 = 10w[j] - 9d$
(2) if $\delta_1 < 0$ then
$z_{j+1} = 0$ and $w[j+1] = 10 \cdot w[j]$
else if $\delta_2 < 0$ then
$z_{j+1} = 1$ and $w[j+1] = \delta_1$
$\vdots$
else if $\delta_9 < 0$ then
$z_{j+1} = 8$ and $w[j+1] = \delta_8$
else
$z_{j+1} = 9$ and $w[j+1] = \delta_9$

Algorithm 1: Pseudocode for type1 digit recurrence division.

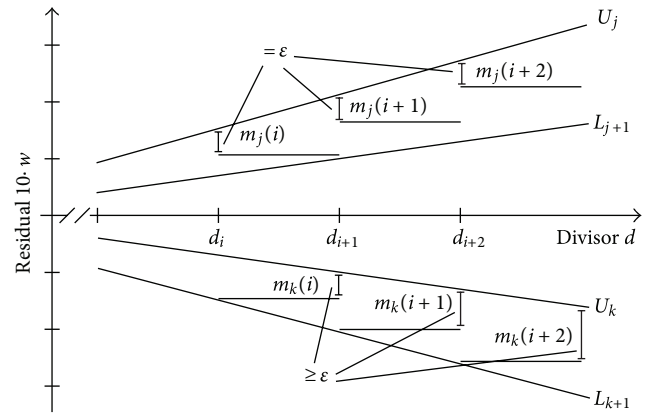

Figure 2: P-D diagram for the type2 QDS function.

These selection constants are bounded by selection intervals $m_k(i) \in [L_k, U_{k-1}]$, with the containment condition [20]

$$L_k(d_i) = (k - \rho)\, d_i, \tag{12a}$$

$$U_k(d_i) = (k + \rho)\, d_i. \tag{12b}$$

Furthermore, the continuity condition states that every value $10w[j]$ must belong at least to one selection interval [20]. Considering the maximum error due to truncation, the continuity condition can be expressed by

$$U_{k-1}(d_i) - L_k(d_{i+1}) \geq \epsilon = 2 \cdot 10^{-t+1}. \tag{13}$$

If we suppose the subrange width to be constant ($d_{i+1} - d_i = 10^{-\delta}$) and consider that the minimum overlapping occurs for minimum $d_{\min}$ and maximum $k_{\max}$, then we obtain the term

$$\begin{aligned} x := {} & (k_{\max} - 1 + \rho)\, d_{\min} \\ & - (k_{\max} - \rho)\left(d_{\min} + 10^{-\delta}\right) \geq 2 \cdot 10^{-t+1}. \end{aligned} \tag{14}$$

We choose $\rho = 8/9$ ($k_{\max} = 8$) and $d_{\min} = 0.4$ that leads to $\delta = 2$ and $t = 2$. Thus, for the divisors's estimation $\widehat{d}$ is required an accuracy of two fractional digits, and for the residual's estimation $\widehat{10w}$ is required an accuracy of one integer and one fractional digit. Furthermore, the divisor must be pre-scaled such that $0.4 \leq d < 1.0$.

The P-D diagram is a visualization technique for designing a quotient digit selection function and for computing the decision boundaries $m_k(i)$. It plots the shifted residual versus the divisor. The P-D diagram for the type2 quotient digit selection function with the selection constants $m_k(i)$ is shown in Figure 2.

The two MSDs of the dividend and residual address the ROM in order to obtain the signed quotient digit, which comprises 5 bits. In order to reduce the ROM's size, the two MSDs of the divisor and the residual are implemented by using a nonredundant radix-2 representation. This leads to an address that is composed of 15 bits: seven bits from the unsigned radix-2 representation of the divisor's two MSDs and eight bits from the signed radix-2 representation of the residual's two MSDs. Hence, the size of the ROM is $2^{15} \times 5$ bits. Unfortunately, the use of radix-2 representation complicates the multiplication by 10, which is required according to (5). Therefore, an additional binary adder is needed:

$$10 \cdot w[j] = 8 \cdot w[j] + 2 \cdot w[j] = w[j] \ll 3 + w[j] \ll 1. \tag{15}$$

The corresponding algorithm for the calculation of the ROM entries can be derived from the P-D diagram and is depicted in Algorithm 2. It should be noted that the radix-2 representations of $\widehat{d}$ and $\widehat{w}$ are truncations of d and w; that is, $\widehat{d} \leq d$ and $\widehat{w} \leq w$.

Once the quotient digit z_{j+1} has been determined, the multiple $z_{j+1} \cdot d$ is subtracted from the current residual to compute the next residual, as stated in (5). The subtracter is a fast redundant BCD-4221 carry-save adder (CSA), as described in [21], except for the two MSDs in wich we apply

(1) compute $L_k(\widehat{d_i})$ and $U_k(\widehat{d_i})$ for each $\widehat{d_i} \in \{0.40, 0.41, \ldots, 0.99\}$ and $k \in \{-8, \ldots, +8\}$ according to (12a) and (12b)
(2) for all $10 \cdot \widehat{w}_n \in \{-8.8, -8.7, \ldots, +8.8\}$ with $|\widehat{w}| \leq \rho \cdot \widehat{d}_{\max} = 0.88$ compute the quotient digits z as follows:
if $10 \cdot \widehat{w}_n \geq U_7 - \epsilon = U_7 - 0.2$ then $z = 8$
else if $10 \cdot \widehat{w}_n \geq U_6 - 0.2$ then $z = 7$
$\vdots$
else if $10 \cdot \widehat{w}_n \geq U_0 - 0.2$ then $z = 1$
else if $10 \cdot \widehat{w}_n \geq L_0$ then $z = 0$
$\vdots$
else if $10 \cdot \widehat{w}_n \geq L_{-8}$ then $z = -8$

Algorithm 2: Algorithm for the calculation of the type2 ROM entries.

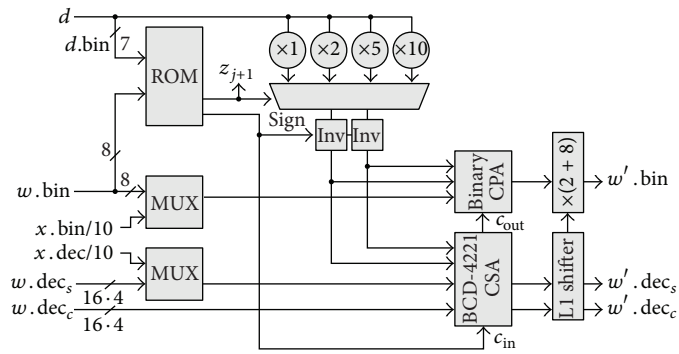

Figure 3: Block diagram of type2 digit recurrence.

(1) Select $z_{j+1} = \mathrm{SEL}_{\mathrm{ROM}}(\widehat{10w[j]}, \hat{d})$
(2) Compute $(w_s[j+1] + w_c[j+1]) =$
$10(w_s[j] + w_c[j]) - z_{j+1} \cdot d$

Algorithm 3: Pseudocode for the type2 digit recurrence.

radix-2 CPAs. Thus, the multiple $z_{j+1} \cdot d$ is composed of two summands:

$$z_{j+1} \cdot d = d^1_{j+1} + d^2_{j+1}, \tag{16}$$

where $d^{1,2} \in \{0, \pm 1d, \pm 2d, \pm 5d, \pm 10d\}$ are precomputed. The multiplies $2d$ and $5d$ can be easily and fast computed by digit recoding and constant shift operation as shown in (9) and (10). The 10's and 2's complements are applied for $d^{1,2} < 0$. For both, radix-2 and BCD-4221 radix-10 representation, the complements can be computed by inverting each bit and adding one. In summary, the subtraction uses a (3 : 1) radix-2 CPA for the two MSDs and a redundant radix-10 (4 : 2) CSA for the remaining digits. The digit recurrence algorithm of the type2 divider is summarized in Algorithm 3, and its block diagram is depicted in Figure 3.

4.4. Type3 QDS Function. The frequency limiting component of the type2 divider is the digit recurrence step, which comprises the ROM (BRAM) to calculate the next quotient digit. This BRAM is slower compared to common FPGA logic. Hence, it appears to be beneficial to remove the slow BRAM from the critical path and use fast FPGA logic instead. To analyze this impact of the BRAM delays, we implement two dividers (type3.a and type3.b), which can also be realized without BRAM in the critical path.

Lang and Nannarelli [14] propose a divider that implements a quotient digit selection function based on CPAs and sign detection. These CPAs subtract fixed values from the current residual with limited precision. These fixed values depend on the estimates of the divisor and are precomputed and stored in a ROM. Furthermore, the quotient digit is divided into two parts, which further simplifies the quotient digit selection function and reduces the critical path. Unfortunately, this divider cannot be implemented efficiently on FPGAs because the required carry-save adder with small error estimation shows a poor performance on the FPGA's slice structure. Nevertheless, to investigate the impact of BRAM delays in the type2 divider, we implement another two dividers (type3.a and type3.b) that have no BRAM in the critical path but a CPA-based quotient digit selection function instead.

The type3 dividers are modifications of the type2 divider. The common architectural features are

(i) the multiples of the divisor are compos of two components: $z_{j+1} \cdot d = d^1_{j+1} + d^2_{j+1}$, where $d^{1,2} \in \{0, \pm 1d, \pm 2d, \pm 5d, \pm 10d\}$,

(ii) the fast redundant BCD-4221 carry-save adders for the digit recurrence step as stated in (5),

(iii) the radix-2 implementation of the two MSDs, and

(iv) the 10's and 2's complements for $d^{1,2} < 0$.

Furthermore, since the architectures of the type2 and type3 dividers are similar, with the exception of the quotient digit selection function, most of the dividers' characteristics are also identical, including the

(i) the P-D diagram,

(ii) the redundancy factor $\rho = 8/9$, $k_{\max} = 8$,

(iii) the need of prescaling the divisor $0.4 \le d < 1.0$,

(iv) the accuracy of the divisor's estimation $\delta = 2$, and

(v) the accuracy of the residual's estimation $t = 2$.

Contrary to the type2 divider, the QDS functions of the type3 dividers are implemented by 16 carry-propagate adders with sign detection. These adders subtract fixed selection constants from the estimation of the current residual. The selection constants are dependent on the current divisor and are stored in a ROM, which is then no more part of the critical path. The selection constants are computed according to Algorithm 4. The common digit recurrence algorithm of the type3.a and type3.b dividers is shown in Algorithm 5, and the corresponding block diagram is depicted in Figure 4.

The sign signals of the binary carry-propagate adders are used to determine the corresponding quotient digit and to select the multiples of the divisor in the digit recurrence

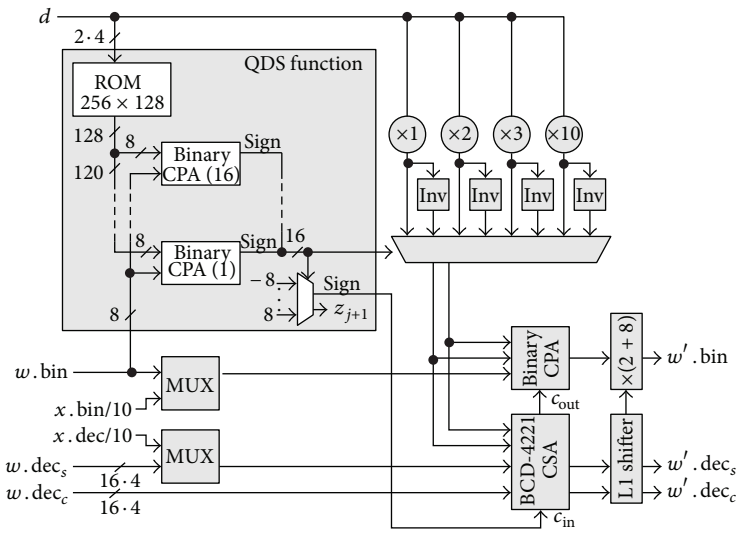

Figure 4: Block diagram of type3 digit recurrence.

(1) compute $L_k(\widehat{d}_i)$ and $U_k(\widehat{d}_i)$ for each $\widehat{d}_i \in \{0.40, 0.41, \ldots, 0.99\}$ and $k \in \{-8, \ldots, +8\}$ according to (12a) and (12b)
(2) compute the selection constants $m_k(\widehat{d}_i)$:

$$m_8\left(\widehat{d}_i\right) = U_7\left(\widehat{d}_i\right) - \epsilon = U_7\left(\widehat{d}_i\right) - 0.2$$
$$m_7\left(\widehat{d}_i\right) = U_6\left(\widehat{d}_i\right) - 0.2$$
$$\vdots$$
$$m_1\left(\widehat{d}_i\right) = U_0\left(\widehat{d}_i\right) - 0.2$$
$$m_0\left(\widehat{d}_i\right) = L_0\left(\widehat{d}_i\right)$$
$$\vdots$$
$$m_{-7}\left(\widehat{d}_i\right) = L_{-7}\left(\widehat{d}_i\right)$$

Algorithm 4: Algorithm for the calculation of the type3 selection constants.

(1) if $m_8\left(\widehat{d}_i\right) - \widehat{10w[j]} < 0$ then $z_{j+1} = 8$
else if $m_7\left(\widehat{d}_i\right) - \widehat{10w[j]} < 0$ then $z_{j+1} = 7$
$\vdots$
else if $m_{-7}\left(\widehat{d}_i\right) - \widehat{10w[j]} < 0$ then $z_{j+1} = -7$
else $z_{j+1} = -8$
(2) Compute $(w_s[j+1] + w_c[j+1]) = 10\left(w_s[j] + w_c[j]\right) - z_{j+1} \cdot d$

Algorithm 5: Pseudocode for the type3 digit recurrence.

iteration step. The advantage of the type3 QDS function is its short critical path, which comprises only one 8-bit binary carry-propagate adder. However, this short critical path is bought at a high price because the complexity is moved to the selection of the divisors multiples. Hence, large fan-in multiplexers with poor latencies are required. In order to minimize the impact of these multiplexers, we designed a new dedicated (17 : 1) multiplexer that exploits the FPGA's fast carry chains and has a delay of only one LUT instance. In the following we name the divider with improved fast multiplexers type3.b and with traditional multiplexers type3.a. The new dedicated (17 : 1) multiplexer uses the 16-sign bits as select lines and is coded as follows:

$$\begin{aligned}
&\text{if sel} = {}'1111.1111.1111.1111' \text{ then } y = x(16)\\
&\text{if sel} = {}'0111.1111.1111.1111' \text{ then } y = x(15)\\
&\qquad\vdots\\
&\text{if sel} = {}'0000.0000.0000.0001' \text{ then } y = x(1)\\
&\text{if sel} = {}'0000.0000.0000.0000' \text{ then } y = x(0),
\end{aligned} \tag{17}$$

with sel(−1) := 1 and sel(16) := 0. In other words, bit $x(i)$ is selected when sel(i) = 0 and sel(i − 1) = 1. The selection of two input signals is implemented in one (5 : 1) LUT, and all signals are combined by a long OR gate that is implemented exploiting the FPGA's fast carry chain. Therefore, one (17 : 1) multiplexer requires nine LUTs, and the type3.b divider with fast multiplexers uses much more LUTs than the type3.a divider (see Section 6). The block diagram of such a (17 : 1) multiplexer is depicted in Figure 5.

4.5. Type4 QDS Function. The type4 divider applies a new algorithm as proposed by us in a previous paper [22]. It is based on the decomposition of the signed quotient digit into three components having values {−5, 0, 5}, {−2, 0, 2}, and {−1, 0, 1} as well as a fast constant digit selection function. In this implementation, neither a ROM for the QDS function nor a multiplexer to select the multiple $z_j \cdot d$ in the digit recurrence iteration step is required. The divider is intended to utilize less resources than other implementations. As

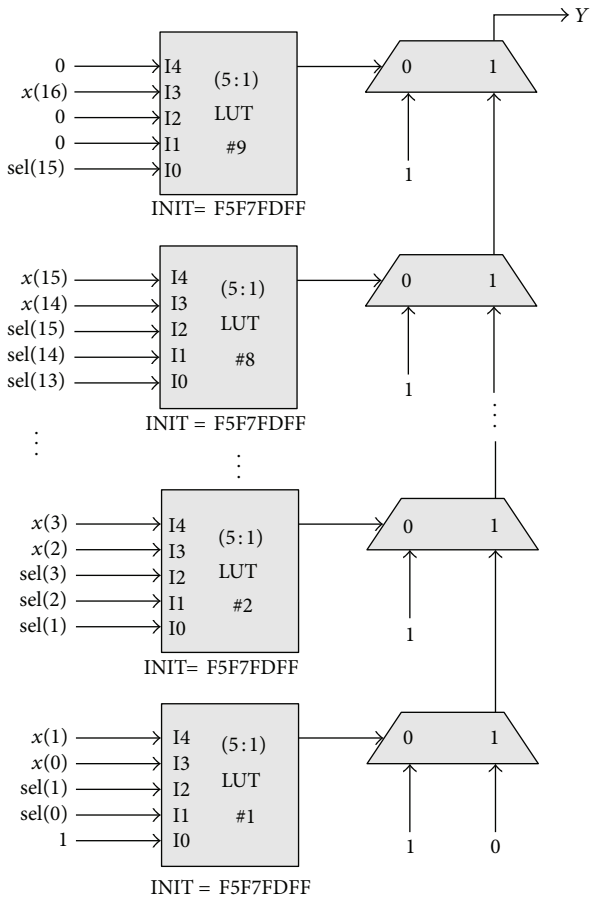

Figure 5: Fast multiplexer.

the type2, type3.a, and type3.b dividers, the type4 divider uses signed-digit redundant quotient calculation, carry-save representation of the residual, fast BCD-4221 CSAs for the digit recurrence, and a radix-2 implementation of the MSDs.

Quotient digits are decomposed into three components $z_{j+1} = 5 \cdot z^3_{j+1} + 2 \cdot z^2_{j+1} + z^1_{j+1}$, and each component z^i is computed by a distinct selection function. These components can hold three values $z^i \in \{-1, 0, 1\}$ so that only two comparators per component are required to distinguish between these values. Since $-8 \le z_{j+1} \le 8$, we get a redundancy factor of $\rho = 8/9$. Furthermore, the selection functions become very simple due to prescaling of the divisor ($0.4 \le d < 0.8$); that is, they are constant functions SEL^i that do not depend on the divisor anymore:

$$z^3_{j+1} = \mathrm{SEL}^3\left(\widehat{10w\,[j]}\right), \tag{18a}$$

$$z^2_{j+1} = \mathrm{SEL}^2\left(\widehat{v^1\,[j]}\right), \tag{18b}$$

$$z^1_{j+1} = \mathrm{SEL}^1\left(\widehat{v^2\,[j]}\right). \tag{18c}$$

The recurrence is then defined as follows:

$$v^1\,[j] = 10w\,[j] - z^3_{j+1} \cdot 5d, \tag{19a}$$

$$v^2\,[j] = v^1\,[j] - z^2_{j+1} \cdot 2d, \tag{19b}$$

$$w\,[j+1] = v^2\,[j] - z^1_{j+1} \cdot d. \tag{19c}$$

The multiples $2d$ and $5d$ can be easily and fast computed according to (9) and (10). Each selection function requires two comparators implemented as carry-save adders that subtract constant values from the residuals' estimations ($\widehat{w[j]}$, $\widehat{v^1[j]}$, and $\widehat{v^2[j]}$). As we will show in the following, these estimations require only the two most significant digits (MSDs) of the corresponding exact values.

First, the selection intervals $[L^i_k, U^i_k]$ for $z^i_{j+1} = k$ with $k \in \{-1, 0, 1\}$ and $i \in \{1, 2, 3\}$ have to be determined, where L^i_k is the smallest and U^i_k is the greatest value of the selection constant $m_k(i)$ such that the next residual is still bounded. Applying the convergence condition (8), the digit recurrence (19a), (19b), and (19c), and the redundancy factor $\rho = 8/9$, we obtain

$$\left|w\,[j]\right| \le \rho d = \frac{8}{9}d, \tag{20a}$$

$$\left|v^2\,[j]\right| \le (\rho + 1)\,d = \frac{17}{9}d, \tag{20b}$$

$$\left|v^1\,[j]\right| \le (\rho + 1 + 2)\,d = \frac{35}{9}d. \tag{20c}$$

From the recurrence $v^1[j] = 10w[j] - z^3_{j+1} \cdot 5d$, $v^1[j] \le (35/9)d$, $z^3_{j+1} \in \{-1, 0, 1\}$, and replacing $10w[j]$ by the upper limit U^3_k, we get

$$U^3_k = (5k + \rho + 3)\,d = \left(5k + \frac{35}{9}\right)d. \tag{21a}$$

Similarly, we obtain the upper limits U^2_k and U^1_k from the recurrence (19b) and (19c), respectively,

$$U^2_k = (2k + \rho + 1)\,d = \left(2k + \frac{17}{9}\right)d, \tag{21b}$$

$$U^1_k = (k + \rho)\,d = \left(k + \frac{8}{9}d\right). \tag{21c}$$

Likewise, the lower limits L^i_k can be computed

$$L^3_k = (5k - \rho - 3)\,d = \left(5k - \frac{35}{9}\right)d, \tag{22a}$$

$$L^2_k = (2k - \rho - 1)\,d = \left(2k - \frac{17}{9}\right)d, \tag{22b}$$

$$L^1_k = (k - \rho)\,d = \left(k - \frac{8}{9}d\right). \tag{22c}$$

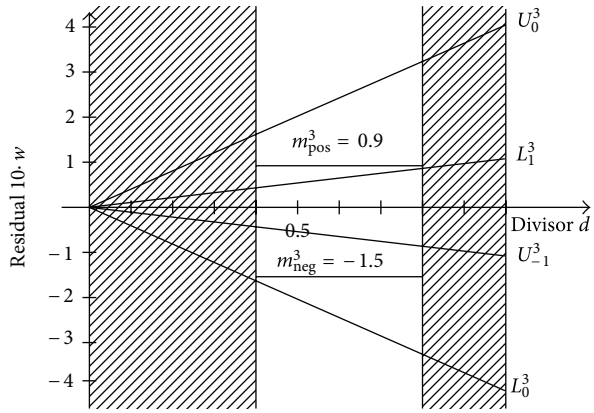

Figure 6: P-D diagram for the type4 QDS function.

Due to the redundancy factor $\rho = 8/9 > 1/2$ we have overlapping regions $[L_{k+1}^i, U_k^i]$, where more than one quotient digit component z_{j+1}^i may be selected. The decision boundary of a selection function should lie inside this overlapping regions. Figure 6 shows the P-D diagram for z^3. The figures for z^2 and z^1 look similar. As z^3 can hold three different values, the selection function requires two decision boundaries. Generally, it is implemented by a staircase function (see Figure 2), but for the bounded divisor $d \in [0.4, 0.8)$ it is independent of the divisor and is reduced to a constant function (see Figure 6). Fortunately, the selection functions for z^2 and z^1 are also independent of the divisor and can also be implemented by constant functions in a similar fashion.

We now determine the required number of fractional digits for the estimates $\widehat{10w}$, $\widehat{v^1}$, and $\widehat{v^2}$. Moreover, we choose suitable selection constants m_{pos}^i and m_{neg}^i. These constants depend on the maximum error due to the estimates and on the minimum overlapping width, which is given by $U_0^i(0.4) - L_1^i(0.8)$ as well as $U_{-1}^i(0.8) - L_0^i(0.4)$. Since we use BCD-4221 carry-save redundant digit representation, for a precision of t digits (one integer digit and $t-1$ fractional digits), we have a maximum error of $\epsilon = 2 \cdot 10^{-(t-1)}$. Moreover, the estimations $\widehat{w}$ are computed by truncations ($w - \widehat{w} \le \epsilon$). This means, for given positive and negative selection constants $m_{\text{pos}}^i > 0$ and $m_{\text{neg}}^i < 0$, the following expressions must be true:

$$U_0^i(0.4) - m_{\text{pos}}^i \ge \epsilon, \qquad L_1^i(0.8) \le m_{\text{pos}}^i, \tag{23}$$

$$L_0^i(0.4) \le m_{\text{neq}}^i, \qquad U_{-1}^i(0.8) - m_{\text{neg}}^i \ge \epsilon. \tag{24}$$

If we choose the selection constants as listed in Table 2, all conditions are fulfilled for a precision of $t = 2$ digits, and each selection function is reduced to two simple 2-digit comparators.

As soon as one component z^i of the quotient digit is computed, the corresponding multiple of the divisor must be subtracted from the partial residual. Each component z^i can hold three different values $z^i \in \{-1, 0, +1\}$ that are multiplied with the corresponding weighted divisor $n \cdot d$ with $n \in \{5, 2, 1\}$. In other words, $n \cdot d$ is either passed, negated, or reset. Passing and resetting the multiple of the divisor is performed at no extra cost. Negation is accomplished by bit inversion and adding +1 through the carry inputs of the carry-save adders in (19a), (19b), and (19c). In summary, this is a very fast operation and requires only two LUTs per digit.

Table 2: Constants for type4 quotient digit selection function.

	m_{pos}^i	m_{neg}^i	$U_0^i(0.4) - m_{\text{pos}}^i$	$U_{-1}^i(0.8) - m_{\text{neg}}^i$
z^3	0.9	−1.5	0.656	0.611
z^2	0.1	−0.7	0.656	0.611
z^1	0.1	−0.3	0.255	0.211

Similar to type2 and type3 division, we implement the two MSDs of the type4 residual by using a nonredundant radix-2 representation, which requires less LUTs and results in faster quotient digit selection function. The pseudocode for the digit recurrence iteration is shown in Algorithm 6, and the corresponding block diagram is depicted in Figure 7.

4.6. Proposed Decimal Fixed-Point Divider. For each type of division algorithms presented in the preceding sections we have implemented a corresponding fixed-point divider, which is described in the following. For all fixed-point dividers we expect the input operands to be normalized.

The block diagram of the type1 divider is depicted in Figure 8. First, the divisor multiples $\{2d, \ldots, 9d\}$ are precomputed. In the following cycles $p = 16$ quotient digits ($16 + 1$ if the first digit is zero) are computed one by one, followed by an additional rounding digit. The quotient (Z) and the quotient +1 (ZP1) are computed on-the-fly by using two registers that are updated every cycle. The algorithm has the advantage that no additional slow decimal CPA is required to compute the incremented quotient. It is described more precisely in Section 5. The normalization of the result is also performed on-the-fly by locking the conversion when 16 + 1 quotient digits for the significand and the rounding digit have been computed; that is, in the worst case, when the first quotient digit is zero, $16 + 1 + 1 = 18$ cycles are required. Furthermore, the sticky bit is calculated, which is required for rounding. It is set to one whenever the final remainder is unequal to zero.

The architectures of the type2, type3.a, type3.b, and type4 are similar in their structure but differ in their digit recurrence algorithm and scaling. The common block diagram is shown in Figure 9. First, the dividend and the divisor are rescaled

$$d \in [0.1, 1.0) \implies d' \in \begin{cases} [0.4, 1.0) & \text{for type2, type3.a/b} \\ [0.4, 0.8) & \text{for type4} \end{cases} \tag{25}$$

and five as well as two times the divisor are precomputed according to (9) and (10). Then, the operands are recoded because the digit recurrence iteration uses a redundant digit representation with BCD-4221 carry-save adders.

The digit recurrence retires in each iteration one signed quotient digit z_k. The on-the-fly-conversion algorithm converts this signed-digit representation to the BCD-8421-coded

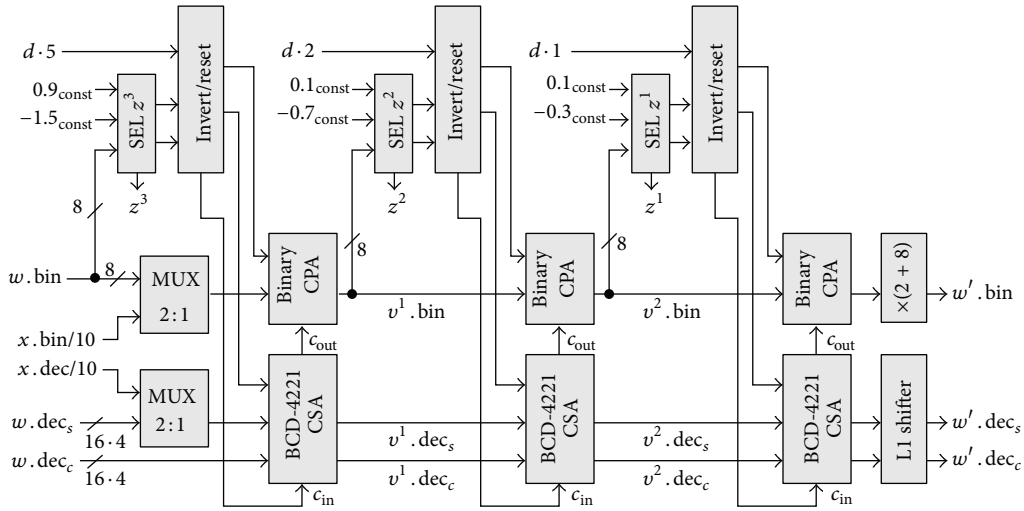

Figure 7: Block diagram of type4 digit recurrence.

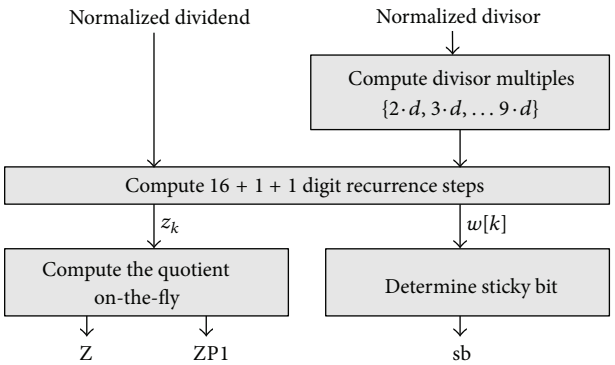

Figure 8: Block diagram of the normalized type1 division.

(1) select

$$z^3_{j+1} = \begin{cases} 1 & \text{if } 0.9 \le \widehat{10w}[j] \\ 0 & \text{if } -1.5 \le \widehat{10w}[j] < 0.9 \\ -1 & \text{if } \widehat{10w}[j] < -1.5 \end{cases}$$

and compute

$$(v^1_s[j] + v^1_c[j]) = 10(w_s[j] + w_c[j]) - z^3_{j+1} \cdot 5d$$

(2) select

$$z^3_{j+1} = \begin{cases} 1 & \text{if } 0.1 \le \widehat{v^1}[j] \\ 0 & \text{if } -0.7 \le \widehat{v^1}[j] < 0.1 \\ -1 & \text{if } \widehat{v^1}[j] < -0.7 \end{cases}$$

and compute

$$(v^2_s[j] + v^2_c[j]) = (v^1_s[j] + v^1_c[j]) - z^2_{j+1} \cdot 2d$$

(3) select

$$z^1_{j+1} = \begin{cases} 1 & \text{if } 0.1 \le \widehat{v^2}[j] \\ 0 & \text{if } -0.3 \le \widehat{v^2}[j] < 0.1 \\ -1 & \text{if } \widehat{v^2}[j] < -0.3 \end{cases}$$

and compute $(w_s[j+1] + w_c[j+1]) = (v^2_s[j] + v^2_c[j]) - z^1_{j+1} d$

Algorithm 6: Pseudo code for the type 4 digit recurrence.

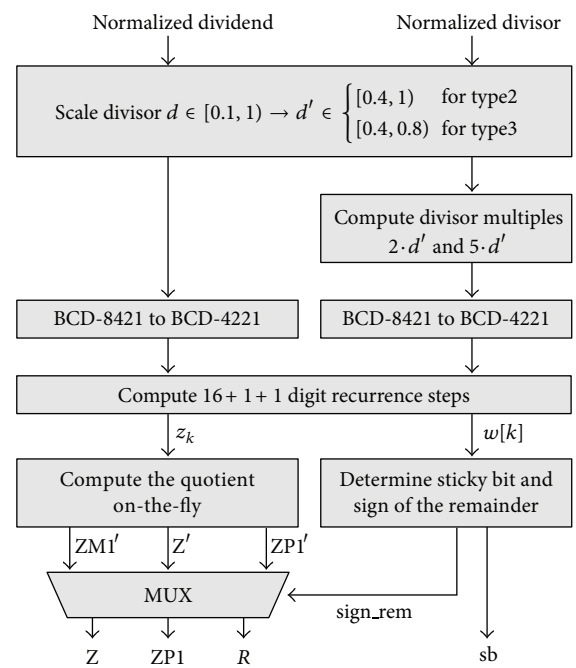

Figure 9: Block diagram of the normalized type2, type3.a, type3.b, and type4 divisions.

result [20] and computes the quotient (Z), the quotient +1 (ZP1), and quotient −1 (ZM1). This conversion is accomplished every iteration step and does not need a slow CPA. The incremented and decremented quotients are required for rounding. Moreover, the quotient is also normalized on-the-fly by locking the conversion when 16 + 1 quotient digits (including the rounding digit) have been computed. The algorithm is described explicitly in Section 5 because it is accomplished together with the gradual underflow handling of the floating-point divider.

```
if (¬qNaN_X ∧ ¬sNaN_X) ∧ (qNaN_D ∨ sNaN_D) then
    C'_X = C_D,    C'_D = 0...01
    q'_X = q_D,    q'_D = 0 + bias = 398
else if (qNaN_X ∨ sNaN_X) then
    C'_X = C_X,    C'_D = 0...01
    q'_X = q_X,    q'_D = 0 + bias = 398
else  // no changes
    C'_X = C_X,    C'_D = C_D
    q'_X = q_X,    q'_D = q_D
```

Algorithm 7: Algorithm of NaN handling.

Furthermore, the sticky bit is calculated, which is required for rounding. It is set to one whenever the final remainder is unequal to zero. Moreover, when the final remainder is negative, the quotient of the type2, type3.a, type3.b, and type4 dividers has to be adjusted by subtracting one LSD. This subtraction does not need another slow CPA because the quotient −1 (ZM1) has already been computed on-the-fly. The calculations of both, the sticky and sign bit, require the reduction of the redundant remainder by a CPA. This CPA might be subdivided into multiple smaller CPAs to keep the latency low.

5. IEEE 754-2008 Floating-Point Division

The IEEE 754-2008 compliant decimal floating-point divider is an extension of the normalized fixed-point divider. The divider presented in this paper supports the interchange format IEEE 754-2008 *decimal64* with DPD encoding, but it can be easily adapted to any other precision and exponent range.

The block diagram of the divider is depicted in Figure 10. The DFP division begins with decoding of the dividend X and divisor D and the extraction of their signs, significands, and exponents. In the following, the significands of the operands X (c_X) and D (c_D) are regarded as integers. Therefore, the corresponding exponents are q_X and q_D.

According to [4], if one of the operands is a signaling NaN (sNAN) or quiet NaN (qNAN), then the result is also a quiet NaN with the payload of the original NaN. Hence, in order to preserve the payload while using an unmodified divider, the NaN handling unit sets the NaN holding operand as the dividend and resets the divisor to $1.0 \cdot 10^0$, as depicted in Algorithm 7.

The fixed-point dividers presented in Section 4 require normalized operands. Therefore, the number of leading zeros for both operands is counted, and the significands are normalized by barrel shifters. Due to performance issues, the leading zeros counter exploit the FPGA's fast carry chains, as proposed in [23].

The exponent of a floating-point division is first estimated by the normalized quotient in fractional representation (QNF) and is updated iteratively in each digit recurrence step. Since most of the decimal floating-point numbers specified by IEEE 754-2008 have multiple representations of the same value, the result of a division might also have more than one correct representations that differ in the exponent. However, to obtain a unique and reproducible result in the case of such an ambiguity, IEEE 754-2008 defines the exponent of the result, which is called preferred exponent (QP).

Both exponents, QNF and QP, are positive integers and are determined as follows:

$$\begin{aligned} \Delta q &:= \left(q_X - q_D\right), \\ \Delta \mathrm{LZ} &:= \left(\mathrm{LZ}_X - \mathrm{LZ}_D\right), \end{aligned} \tag{26}$$

$$\begin{aligned} \mathrm{QNF} &= \Delta q - \Delta \mathrm{LZ} + \mathrm{offset1}, \\ \mathrm{QP} &= \Delta q + \mathrm{offset1}. \end{aligned} \tag{27}$$

The *offset1* is a bias and assures the quotients to be positive since unsigned integer calculation is used in this paper. The bias is composed of

$$\begin{aligned} \mathrm{offset1} &= 783 \\ &= 398 \;//\; \text{IEEE754-2008 bias} \\ &+ 369 \;//\; q_{\max} \\ &+ 15 \;//\; \text{conversion fractional to integer} \\ &+ 1 \;//\; \text{normalization of the result.} \end{aligned} \tag{28}$$

The normalized fixed-point division unit then computes $p = 16$ significand digits and one additional rounding digit. Additionally, the divider encompasses the on-the-fly conversion unit that also detects and handles gradual underflow with zero delay overhead. One signed quotient digit z_j is retired in each cycle, and the on-the-fly-conversion algorithm converts this signed-digit representation into the BCD-8421-coded partial result [20]. The conversion is accomplished every iteration step and does not need a slow CPA (see Algorithm 8). The partial quotient is stored in a register Z′. Moreover, two additional registers ZM1′ and ZP1′ are provided. ZM1′ is the partial quotient decremented by one least significant digit (LSD), and ZP1′ is the partial quotient incremented by one LSD. ZM1′ is selected when the final residual in the last iteration is negative, and ZP1′ is required by the following rounding operation. Furthermore, the exponent for the normalized significand in integer representation (QNI) is calculated (see (29)), and gradual underflow handling is performed at no extra cost (see Algorithm 8). The exponent QNI is computed by decrementing the exponent of the normalized significand in fractional representation (QNF) in each iteration step by one. The recurrence iteration terminates earlier in case of gradual underflow. The gradual underflow signal (GUF) is asserted *high* when QNI = $q_{\min}$, and the calculated integer quotient is less than 10^p:

$$\mathrm{QNI} = \begin{cases} q_{\min} & \text{if GUF} = 1 \\ \mathrm{QNF} - (p-1) & \text{if } z_1 > 0 \\ \mathrm{QNF} - p & \text{if } z_1 = 0. \end{cases} \tag{29}$$

The extended quotient (composed of the calculated quotient and the rounding digit) must be decremented by one if the

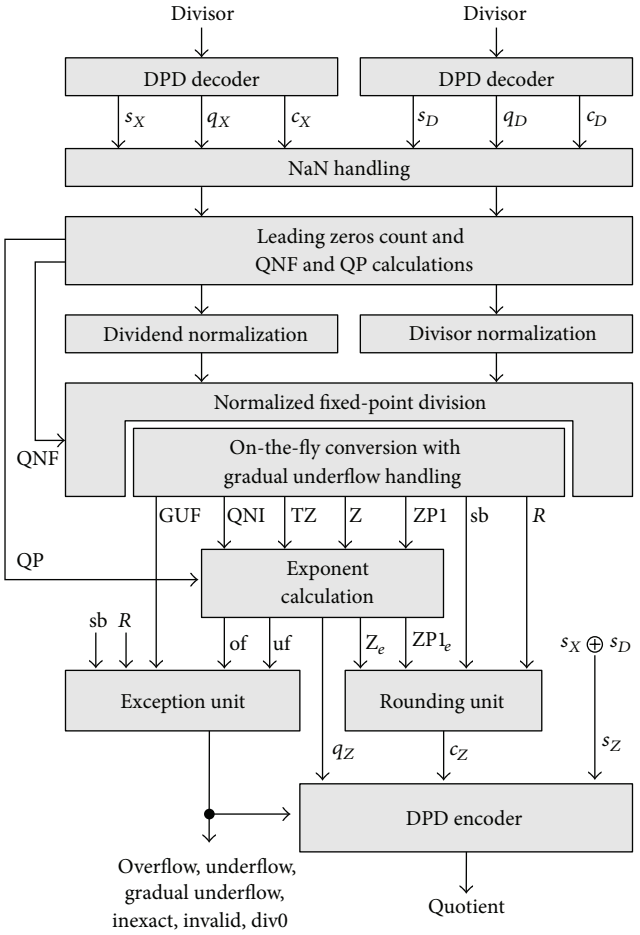

Figure 10: Block diagram of the floating-point divider.

$$
\begin{aligned}
&\mathrm{QNI}' = \mathrm{QNF} + 1\\
&\mathrm{Z}' = \mathrm{ZM1}' = \mathrm{ZP1}' = 0\\
&\mathrm{TZ}' = 0,\ j = 0\\
&\text{while } (\mathrm{Z}' < 10^{p}) \text{ and } (\mathrm{QNI}' \geq q_{\min}) \ \{\\
&\quad \mathrm{Z}' = \begin{cases} 10 \cdot \mathrm{Z}' + z_{j+1} & \text{if } z_{j+1} \geq 0 \\ 10 \cdot \mathrm{ZM1}' + (z_{j+1} + 10) & \text{if } z_{j+1} < 0 \end{cases}\\
&\quad \mathrm{ZM1}' = \begin{cases} 10 \cdot \mathrm{Z}' + \left(z_{j+1} - 1\right) & \text{if } z_{j+1} > 0 \\ 10 \cdot \mathrm{ZM1}' + (z_{j+1} + 9) & \text{if } z_{j+1} \leq 0 \end{cases}\\
&\quad \mathrm{ZP1}' = \begin{cases} 10 \cdot \mathrm{Z}' + \left(z_{j+1} + 1\right) & \text{if } z_{j+1} \geq -1 \\ 10 \cdot \mathrm{ZM1}' + (z_{j+1} + 11) & \text{if } z_{j+1} < -1 \end{cases}\\
&\quad \mathrm{TZ}' = \begin{cases} \mathrm{TZ}' + 1 & \text{if } z_{j+1} = 0 \\ 0 & \text{else} \end{cases}\\
&\quad \mathrm{QNI}' = \mathrm{QNI}' - 1\\
&\quad j = j + 1\\
&\}\\
&\mathrm{QNI} = \mathrm{QNI}' + 1 \text{ // remove the impact of the rounding digit}\\
&R' = \begin{cases} z_{j+1} - 1 & \text{if remainder} < 0 \\ z_{j+1} & \text{else} \end{cases}\\
&\mathrm{ZM1}'' = \lfloor \mathrm{ZM1}' \cdot 10^{-1} \rfloor\\
&\mathrm{Z}'' = \lfloor \mathrm{Z}' \cdot 10^{-1} \rfloor\\
&\mathrm{ZP}'' = \lfloor \mathrm{ZP1}' \cdot 10^{-1} \rfloor\\
&(\mathrm{Z}, \mathrm{ZP1}, R) = \begin{cases} (\mathrm{ZM1}'', \mathrm{Z}'', R' + 10) & \text{if } R' < 0 \\ (\mathrm{Z}'', \mathrm{ZP1}'', R') & \text{else} \end{cases}\\
&\mathrm{sb} = \begin{cases} 0 & \text{if remainder} = 0 \\ 1 & \text{else} \end{cases}\\
&\mathrm{GUF} = \begin{cases} 1 & \text{if } \mathrm{QNI} = q_{\min} \text{ and } \mathrm{Z}' < 10^{p} \\ & \text{and } (\mathrm{sb} = 1 \text{ or } R \neq 0) \\ 0 & \text{else} \end{cases}\\
&\text{if GUF} = 1 \text{ then underflow!}\\
&\mathrm{TZ} = \begin{cases} 0 & \text{if sb} = 1 \\ \mathrm{TZ}' & \text{else} \end{cases}
\end{aligned}
$$

Algorithm 8: On-the-fly conversion with gradual underflow handling.

final remainder is less than zero. The sticky bit (sb) is used for rounding and is asserted *high* whenever the remainder is unequal to zero.

Moreover, the number of trailing zeros (TZ) is computed on-the-fly. This number indicates by how many digits the computed quotient might be shifted to the right without loosing accuracy. The number of trailing zeros has to be counted only for Z' because if $\mathrm{ZM1}'$ is selected, the residual will be unequal to zero and TZ = 0 anyway. The number of trailing zeros is used for the selection between the preferred exponent (QP) and the normalized exponent for integer representation (QNI), as described in the following paragraph.

The exponent calculation unit selects either QP or QNI. The preferred exponent should be selected whenever possible. This is only feasible when QNI < QP ≤ QNI + TZ; that is, there is a sufficient number of trailing zeros to shift the significand to the right. Furthermore, the final exponent must satisfy the minimum and maximum exponent range. The algorithm of the exponent calculation is illustrated in Algorithm 9, where the value

$$\mathrm{offset2} = 783 - \mathrm{bias} = 783 - 398 = 385 \tag{30}$$

is used to add the IEEE 754-2008 bias and to remove *offset1* again (which was introduced in (27)).

Rounding is required when the number of significant digits of the quotient exceeds the length p of a decimal word. The divider presented in this paper provides the five rounding modes as requested by [4]. Rounding either selects the integer quotient Z_e or the incremented quotient $\mathrm{ZP1}_e = \mathrm{Z}_e + 1$. As proven by Theorem A.1 in the appendix, rounding overflow cannot occur in DFP division. Rounding overflow would occur, if $\mathrm{Z}_e + 1$ overflows due to rounding. The selection of the quotient depends on the rounding mode, the rounding digit (R), the sticky bit (sb), and the least significant digit (LSD) of the quotient. The calculation of the round up detection is summarized in Table 3.

Finally, the result is encoded again, and the exception unit might assert six exception signals. These are division by zero, invalid operation, result is infinite, inexact, overflow, and underflow. Division by zero is asserted when the divisor equals zero but the dividend is unequal to zero. The operation is called invalid when both operands are zero, both operands are infinity, or any of the operands is either a signaling or a quiet NaN. The inexact flag is asserted when the

Table 3: Round-up detection.

Rounding mode	Round-up detection
Round Ties To Even	$(R > 5) + (R \equiv 5) \cdot \text{sb}$ $+(R \equiv 5) \cdot \overline{\text{sb}} \cdot \text{LSD}(0)$
Round Ties To Away	$(R \geq 5)$
Round Toward Positive	$\overline{\text{sign}} \cdot ((R > 0) + \text{sb})$
Round Toward Negative	$\text{sign} \cdot ((R > 0) + \text{sb})$
Round Toward Zero	0

Legend: "+": logical OR, "·": logical AND.

offset2 = 783 − 398 = 385
if (QP ≤ QNI) or (QNI + TZ < QP)
or (QP > $q_{\max}$) then
(a) QNI > $q_{\max}$ ⇒ overflow!
(b) $q_{\min}$ ≤ QNI ≤ $q_{\max}$ ⇒
q_Z = QNI − offset2
Z_e = Z
$ZP1'_e$ = ZP1
(c) QNI < $q_{\min}$ ⇒ underflow!
else
(a) $q_{\min}$ ≤ QP ≤ $q_{\max}$ ⇒
q_Z = QP − offset2
Z_e = Z ≫ (QP − QNI)
$ZP1_e$ = ZP1 ≫ (QP − QNI)
(b) QNI < $q_{\min}$ ⇒ underflow!

Algorithm 9: Exponent calculation.

rounding digit or the sticky bit is unequal to zero. Overflow and underflow are computed as described in the exponent calculation unit and are listed in Algorithms 8 and 9. The infinity exception is asserted when the result is greater than the largest number, that is, when either overflow or division by zero exception is asserted, or when the dividend is infinity while the divisor is a finite number.

6. Implementation Results

All dividers are modeled using VHDL and are implemented for Xilinx Virtex-5 devices with speed grade −2 using Xilinx ISE 10.1. The postplace and route results of the fixed-point dividers are listed in Table 4.

All five types of fixed-point dividers require 19 cycles to perform a division. This includes 17 cycles to determine the quotient digits and the rounding digit, one additional cycle to normalize the result (when the first quotient digit is zero), and one cycle latency to perform on-the-fly conversion and sticky bit determination. As expected, the type1 divider uses the most resources in terms of look-up tables (LUTs) and flip-flops (FFs). The type2, type3.a, and type3.b divider consumes less LUTs but require, further five (type2) or two (type3.a and type3.b) BRAMs. However, comparing the delay of the three dividers leads to an unexpected result. Contrary to decimal divider implementations in ASIC designs, on FPGA platforms the shift-and-subtract algorithm is the fastest. The reason is that the signal propagation on the FPGA's internal fast carry chains is faster than interconnections between slices over the FPGA's general routing matrix. The type1 divider exploits this fast carry chains, whereas type2, type3.a, type3.b, and type4 dividers only use the normal, slow slice interconnection resources. In ASIC implementations, for instance, the longest paths of the type4 divider would be much shorter than the longest path of the type1 divider, but on FPGA architectures the situation is the opposite.

One of the fastest decimal fixed-point divider on ASICs is the design of Lang and Nannarelli [14]. They minimize the critical path in the digit recurrence by using a fast quotient digit selection function based on binary carry-propagate adders that subtract fixed values from the estimation of the current residual. These fixed values are dependent on estimations of the divisor and are precomputed and stored in a ROM. Therefore, the latency of the ROM does not contribute to the critical path of the digit recurrence. The implementation of such a divider on FPGAs would have a poor performance because the required carry-save adder with small error estimation cannot be implemented efficiently on the FPGA's slice structure. However, the concept of removing the ROM from the critical path is applied to the type3.a and type3.b dividers, and the corresponding postplace and route results are shown in Table 4. These results point out that the type3.a utilizes a similar amount of LUTs as the type2 divider but the cycle time is much higher. The reason for this increased cycle time can be explained by the raised complexity of the multiplexers for the selection of the divisor's multiples in the digit recurrence step. These multiplexers show a poor performance in terms of propagation delay because they are implemented as a tree with slow slice interconnections. On the contrary, the type3.b divider applies dedicated fast multiplexers, which have a propagation delay of only one LUT instance and eight fast carry chains. This dedicated fast multiplexer speeds up the divider and reduces the maximum cycle time by one nanosecond. Unfortunately, the number of used LUTs is increased dramatically by approximately 1000 LUTs.

The comparison of the type2, type3.a, type3.b, and type4 dividers in terms of speed and area (number of occupied LUTs and FFs combined) shows that the type2 algorithm is the fastest and requires the least number of slices. Hence, there is no benefit in removing the ROM's latency from the critical path because the complexity is moved to the selection of the divisor's multiples. The type1 divider is only 5% faster than the type2 divider, but the speed is bought at a high price because the number of occupied slices is 75% higher. Therefore, the type2 divider shows a good tradeoff in terms of area and latency and is used for the floating-point divider presented in this paper. In the following paragraph we compare the type2 implementation with other published fixed-point dividers.

Four other FPGA-based dividers are presented by Ercegovac and McIlhenny [17, 18], Deschamp and Sutter [24], Zhang et al. [25], and Véstias and Neto [12]. These implementations are compared to our type2 divider in the following. The divider presented in [17, 18] is based on a digit recurrence algorithm that only requires limited-precision multipliers,

Table 4: Results for normalized decimal fixed-point dividers, $p = 16 + 1$.

	Type1	Type2	Type3.a	Type3.b	Type4
Number of LUTs	3595	1704	1749	2751	1846
Number of FFs	1234	1126	1262	1262	1031
Number of LUTs and FFs combined	3868	2210	2304	3240	2203
Number of 36 k BRAM	0	5	2	2	0
Number of LUTs normalized to the type2 divider	2.11	1.00	1.03	1.61	1.08
Number of FFs normalized to the type2 divider	1.10	1.00	1.12	1.12	0.92
Number of LUTs and FFs combined normalized to the type2 divider	1.75	1,00	1.04	1.47	1.00
Cycle time (ns)	8.1	8.5	10.1	9.1	12.1
Overall latency (ns) (19 × cycle time)	154	162	192	173	230
Max. frequency (MHz)	123	118	99	110	83
Max. frequency normalized to the type2 divider	1.05	1.00	0.84	0.93	0.70

Table 5: Performance comparison of fixed-point dividers.

	Occupied area	Cycle time (ns)	Latency (ns)
Ercegovac and McIlhenny [17] ($p = 14$, Virtex-5)	1263 (6 : 2) LUTs	13.1	197
Type2 divider, equalized to $p = 14$	1692 (6 : 2) LUTs	8.5	136
Deschamps and Sutter [24] ($p = 16$, Virtex-4)			
(i) Nonrestoring algorithm	2974 (4 : 1) LUTs	21.4	386
(ii) SRT-like algorithm	3799 (4 : 1) LUTs	16.6	300
Zhang et al. [25] ($p = 16$, Virtex-II Pro)	3976 slices	20.0	420
Véstias and Neto [12] ($p = 16$, Virtex-4, Newton-Raphson)			
(i) Type A3 (0 DSPs, 118 cycles)	2756 (4 : 1) LUTs	3.4	394
(ii) Type A4 (0 DSPs, 90 cycles)	3768 (4 : 1) LUTs	3.4	306
(iii) Type A5 (7 DSPs, 112 cycles)	2091 (4 : 1) LUTs	3.4	380
(iv) Type A6 (10 DSPs, 86 cycles)	2718 (4 : 1) LUTs	3.4	292
Type2 divider, equalized to $p = 16$	1704 (6 : 2) LUTs	8.5	153

adders, and LUTs. Furthermore, a compensation term is computed in the digit recurrence that compensates the error caused by this limited precision. The design is optimized for Virtex-5 FPGAs and has a good area characteristic (a 14-digit divider requires 1263 LUTs) but suffers from a high cycle time of 13.1 ns, which is more than 50% higher compared to the type1 or type2 design proposed in this paper. The reason for that high cycle time is (similar to the type4 divider) the long critical paths across many LUTs that cannot exploit the FPGA's fast carry chains. Therefore, the design would probably fit better on ASICs with dedicated routing. The dividers implemented in [24] apply two different digit recurrence algorithms with a redundancy of $\rho = 1$. The quotient digit selection function is very complex because it requires three MSDs of the divisor and four MSDs of the residual. The cycle times are high but it must be taken into account that these designs are optimized for Virtex-4 FPGAs. Nevertheless, the number of used LUTs is very high. The divider presented in [25] applies a radix-100 digit recurrence algorithm with comprehensive prescaling of the dividend and divisor. It is optimized for Virtex-II Pro devices; hence, it is hard to compare it with the dividers presented in this paper. However, the critical path includes a decimal carry-save adder tree, two carry-propagate adder stages, and three multiplexer stages. It can therefore be assumed that this algorithm would also have a poor performance on Virtex-5 devices, although two quotient digits are retired in each iteration step. Véstias and Neto [12] propose four different decimal dividers optimized for different speed and area tradeoffs. Contrary to the circuits presented in this paper, the dividers apply the Newton-Raphson algorithm. The used multiplier is sequential; therefore, many cycles per division are required. However, each divider has a higher delay and a greater LUT usage compared to our type2 implementation.

Table 5 lists the results of the dividers presented in [17, 24, 25] together with the results of the type2 divider. However, for a fair performance comparison the precision of the type2 divider is equalized to match the number of calculated quotient digits of the corresponding divider. Furthermore it should be noted that the dividers presented in [12, 17, 24, 25] do not provide rounding support.

For the floating-point implementation, the fixed-point divider of type2 is used because type1 and type3.b consume too many resources, and type3.a and type4 are too slow. However, five additional BRAMs are required, but these are available in sufficient quantities on Virtex-5 devices. The corresponding postplace and route results for two configurations with different numbers of serialization stages are listed in Table 6. A comparison with other implementations is

Table 6: Postplace and route result of the floating-point divider based on type2.

	Config. 1	Config. 2
Number of cycles per float division	23	21
Number of LUTs	3231	3205
Number of FFs	1630	1377
Number of LUTs and FFs combined	3571	3566
Cycle time (ns)	8.5	9.1
Overall latency (ns)	195.5	191
Max. frequency (MHz)	118	110

Table 7: Postplace and route result of a 64-bit binary floating-point divider (Core Gen).

	Cycles per division			
	1	10	20	55
No. of LUTs	3161	592	425	394
No. of FFs	196	518	469	542
No. of LUTs and FFs comb.	3232	917	675	675
Cycle time (ns)	153.7	20.8	8.9	6.8
Overall latency (ns)	153.7	208	178	374
Max. frequency (MHz)	6.5	48	112	147

complicated because there are no other FPGA-based implementations of decimal floating-point dividers published yet. Therefore, we can only compare the decimal divider with binary dividers implemented on the same FPGA. In Table 7 postplace and route results of binary floating-point dividers for the double data format on a Virtex-5 provided by the Xilinx Core Generator [26] are listed. The dividers also provide exception signals *overflow*, *underflow*, *invalid operation*, and *divide-by-zero*. Unfortunately, the binary floating-point divider does not support gradual underflow because this feature increases the complexity. The divider with one cycle per division is an unpipelined and fully parallel design, and the divider that requires 55 cycles per division is a fully sequential design. The binary floating-point divider with 20 cycles per division is best suited for a comparison with the decimal floating-point divider because it requires a comparable number of cycles for one operation. Obviously, decimal arithmetic has a great overhead regarding the total number of used LUTs, but the cycle time and latency are similar. This resource overhead can be explained, on the one hand, by the inefficient representation of decimal values on binary logic and, on the other hand by the more complex specification of the decimal standard IEEE 754-2008. For instance, decimal floating-point division requires additional encoders, decoders, and a normalization stage.

7. Conclusion

In this paper we have presented five different radix-10 digit recurrence division algorithms. The type1 divider implements the simple shift-and-subtract algorithm. The type2 divider is based on a nonrestoring algorithm with ROM-based quotient digit selection function. The type3.a and type3.b dividers are both similar to type2 but use fast binary carry-propagate adders in the quotient digit selection function. However, the type3.a divider uses LUT-based multiplexers whereas the type3.b uses fast carry chain-based multiplexers. The type4 divider applies a new algorithm with constant digit selection functions. This type4 divider requires neither a ROM nor multiplexers to select multiples of the divisor.

We have shown that the subtract-and-shift algorithm with the worst latency in ASIC architectures is the fastest design on FPGAs, but uses the most FPGA resources in terms of LUTs. A good tradeoff between latency and area is the architecture type2 with a redundant carry-save representation of the residual, a radix-2 representation of the two MSDs, and a quotient digit selection function implemented in ROM. Furthermore, we have extended thisfixed-point to a fully IEEE 754-2008 compliant decimal floating-point divider for decimal64 data format, and, finally, we have shown implementation results of all dividers.

Appendix

Proofs

Theorem A.1 (rounding overflow). *Let us consider a decimal floating-point division $Q = X/D$ with the signs s_i, the significands c_i, the exponents q_i, and the precision p*

$$\left(-1^{s_Q} \cdot c_Q \cdot 10^{q_Q}\right) = \frac{\left(-1^{s_X} \cdot c_X \cdot 10^{q_X}\right)}{\left(-1^{s_D} \cdot c_D \cdot 10^{q_D}\right)}. \tag{A.1}$$

Then, rounding overflow, that is, an overflow that arises due to adding +1 to the least significant digit of the final result, cannot occur.

Proof. Assume to the contrary that there are a floating-point dividend X, a floating-point divisor D, and a proper rounding mode, where the decimal floating-point division produces a rounding overflow. Without loss of generality we consider in the following only positive operands ($s_X = s_D = 0$), normalized significands ($c_X, c_D \geq 10^{p-1}$), and the exponents to be zero ($q_X = q_D = 0$). In the case of rounding overflow, the p digits of the quotient must be all nines, and the remainder must be unequal to zero, that is,

$$\underbrace{c_X}_{p \text{ digits}} = c_Q \cdot c_D + \text{rem} = \underbrace{9.9\cdots9}_{p \text{ digits}} \cdot \underbrace{c_D}_{p \text{ digits}} + \underbrace{\text{rem}}_{p \text{ digits}}, \tag{A.2}$$

$$\text{with } 0 < \text{rem} < c_D \cdot 10^{-p+1}. \tag{A.3}$$

Condition (A.2) is fulfilled for

$$c_D = 10^{p-1}, \qquad c_X = \underbrace{9.9\cdots9}_{p \text{ digits}}, \tag{A.4}$$

but this violates condition (A.3) because the remainder is zero (rem = 0); that is, the result is exact, no round-up is applied, and hence no rounding overflow occurs.

Otherwise, if the divisor is greater than 10^{p-1} it follows that

$$\begin{aligned}\underbrace{c_X}_{p \text{ digits}} &= (9.9\cdots9)\cdot c_D + \text{rem}\\ &= \left(10 - 10^{-p+1}\right)\cdot c_D + \text{rem}\\ &= 10\cdot c_D + \underbrace{\left(-10^{-p+1}\cdot c_D + \text{rem}\right)}_{<0}\\ &= \underbrace{10\cdot c_D + \mu}_{\geqslant p+1 \text{ digits}}\end{aligned} \tag{A.5}$$

with $-10 < \mu < 0$. The left term of the equation has, by definition, a precision of p digits while the right term of the equation has a precision of more than p digits. This contradiction, finally, proves that no rounding overflow can occur. □

References

[1] M. Baesler, S. Voigt, and T. Teufel, "FPGA implementations of radix-10 digit recurrence fixed-point and floating-point dividers," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '11)*, pp. 13–19, IEEE Computer Society, Los Alamitos, CA, USA, December 2011.

[2] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating- Point Arithmetic*. New York, NY, USA, August 1985.

[3] M. F. Cowlishaw, "Decimal floating-point: algorism for computers," in *Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH '03)*, pp. 104–111, IEEE Computer Society, Washington, DC, USA, June 2003.

[4] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. New York, NY, USA, August 2008.

[5] ANSI/IEEE. *ANSI/IEEE Std 854-1987: An American National Standard: IEEE Standard for Radix-Independent Floating-Point Arithmetic*. New York, NY, USA, October 1987.

[6] A. Y. Duale, M. H. Decker, H. G. Zipperer, M. Aharoni, and T. J. Bohizic, "Decimal floating-point in z9: an implementation and testing perspective," *IBM Journal of Research and Development*, vol. 51, no. 1-2, pp. 217–227, 2007.

[7] C. F. Webb, "IBM z10: the next-generation mainframe microprocessor," *IEEE Micro*, vol. 28, no. 2, pp. 19–29, 2008.

[8] L. Eisen, J. W. Ward, H. W. Tast et al., "IBM POWER6 accelerators: VMX and DFU," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 663–683, 2007.

[9] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd, "Power7: IBM's next-generation server processor," *IEEE Micro*, vol. 30, no. 2, pp. 7–15, 2010.

[10] S. F. Oberman and M. J. Flynn, "Division algorithms and implementations," *IEEE Transactions on Computers*, vol. 46, no. 8, pp. 833–854, 1997.

[11] L. K. Wang and M. J. Schulte, "A decimal floating-point divider using newton-raphson iteration," *Journal of VLSI Signal Processing Systems*, vol. 49, no. 1, pp. 3–18, 2007.

[12] M. Véstias and H. Neto, "Revisiting the newton-raphson iterative method for decimal divisionpages," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '11)*, pp. 138–143, IEEE Computer Society Press, September 2011.

[13] H. Nikmehr, B. Phillips, and C. C. Lim, "Fast decimal floating-point division," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 9, pp. 951–961, 2006.

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

[15] A. Vázquez, E. Antelo, and P. Montuschi, "A radix-10 SRT divider based on alternative BCD codings," in *Proceedings of the 25th IEEE International Conference on Computer Design (ICCD '07)*, pp. 280–287, IEEE Computer Society Press, Los Alamitos, CA, USA, October 2007.

[16] E. Schwarz and S. Carlough, "Power6 decimal dividepages," in *Proceedings of the 18th IEEE International Conference on Application-Specific Systems Architectures and Processors (ASAP '07)*, pp. 128–133, IEEE Computer Society, July 2007.

[17] M. D. Ercegovac and R. McIlhenny, "Design and FPGA implementation of radix-10 algorithm for division with limited precision primitives," in *Proceedings of the 42nd Asilomar Conference on Signals, Systems and Computers (ASILOMAR '08)*, pp. 762–766, IEEE Computer Society, Pacific Grove, Calif, USA, October 2008.

[18] M. D. Ercegovac and R. McIlhenny, "Design and FPGA implementation of radix-10 combined division/square root algorithm with limited precision primitives," in *Proceedings of the 44th Asilomar Conference on Signals, Systems and Computers (Asilomar '10)*, pp. 87–91, IEEE Computer Society, Pacific Grove, Calif, USA, November 2010.

[19] F. Y. Busaba, C. A. Krygowski, W. H. Li, E. M. Schwarz, and S. R. Carlough, "The IBM z900 decimal arithmetic unit," in *Proceedings of the 35th Asilomar Conference on Signals, Systems and Computers*, vol. 2, pp. 1335–1339, IEEE Computer Society, November 2001.

[20] M. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*, Kluwer Academic Publishers, Norwell, Mass, USA, 1994.

[21] M. Baesler, S. O. Voigt, and T. Teufel, "A decimal floating-point accurate scalar product unit with a parallel fixed-point multiplier on a Virtex-5 FPGA," *International Journal of Reconfigurable Computing*, vol. 2010, Article ID 357839, 13 pages, 2010.

[22] M. Baesler, S. O. Voigt, and T. Teufel, "A radix-10 digit recurrence division unit with a constant digit selection function," in *Proceedings of the 28th IEEE International Conference on Computer Design (ICCD '10)*, pp. 241–246, IEEE Computer Society, Los Alamitos, CA, USA, October 2010.

[23] M. Baesler, S. O. Voigt, and T. Teufel, "An IEEE 754-2008 decimal parallel and pipelined FPGA floating-point multiplier," in *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL '10)*, pp. 489–495, IEEE Computer Society, Washington, DC, USA, September 2010.

[24] J. P. Deschamps and G. Sutter, "Decimal division: algorithms and FPGA implementations," in *Proceedings of the 6th Southern Programmable Logic Conference (SPL '10)*, pp. 67–72, IEEE Computer Society, March 2010.

[25] Y. Zhang, D. Chen, L. Chen et al., "Design and implementation of a readix-100 decimal division," in *Proceedings of IEEE Symposium on Circuit and System (ISCAS '09)*, Taibei, Taiwan, May 2009.

[26] Xilinx. Xilinx LogiCORE Floating-Point Operator v4.0, April 2008.