Hardware division by small integer constants

Fatih Ugurdag, Florent de Dinechin, Yilmaz Serhan Gener, Sezer Gören, Laurent-Stéphane Didier

Hardware division by small integer constants

H. Fatih Ugurdag, F. de Dinechin, Yılmaz Serhan Gener,
Sezer Gören, Laurent-Stéphane Didier

Abstract—This article studies the design of custom circuits for division by a small positive constant. Such circuits can be useful to specific FPGA and ASIC applications. The first problem studied is the Euclidean division of an unsigned integer by a constant, computing a quotient and a remainder. Several new solutions are proposed and compared against the state of the art. As the proposed solutions use small look-up tables, they match well the hardware resources of an FPGA. The article then studies whether the division by the product of two constants is better implemented as two successive dividers or as one atomic divider. It also considers the case when only a quotient or only a remainder are needed. Finally, it addresses the correct rounding of the division of a floating-point number by a small integer constant. All these solutions, and the previous state of the art, are compared in terms of timing, area, and area-timing product. In general, the relevance domains of the various techniques are very different on FPGA and on ASIC.

Index Terms—Integer constant division, IP core generation, parameterized HDL generator, low latency combinational circuit, FPGA synthesis, ASIC synthesis.

1 Introduction

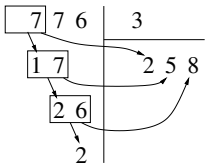
This article considers division by a small integer constant and demonstrates operators for it that can be more efficient than approaches based on standard division [1] or on multiplication by the inverse [2], [3], [4].

1.1 Motivation

Division by a small integer constant is an operation that occurs often enough to justify investigating a specific operator for it. For instance, the core of the Jacobi stencil algorithm computes the average of 3 values: this involves a division by 3. Small integer constants are quite common in such situations. Division by 5 also occurs in decimal-binary conversions. Hardware Euclidean division by a small integer constant can also be used to interleave memory banks in numbers that are not powers of two: if we have D memory banks, an address A must be translated to address A/D in bank $A \mod D$. Finally, a motivation of the present work is the emerging field of High Level Synthesis, which studies the compilation into hardware of applications written in classical sequential languages. There, division by constants will happen in all sort of situations, and it is interesting to provide optimized architectures in such cases.

- *H.F. Ugurdag is with the Department of Electrical and Electronics Engineering, Ozyegin University, Turkey, 34794. E-mail: fatih.ugurdag@ozyegin.edu.tr*
- *F. de Dinechin is with Institut National des Sciences Appliquées, Lyon, France. E-mail:Florent.de-Dinechin@insa-lyon.fr*
- *Y.S. Gener and S. Gören are with the Department of Computer Engineering, Yeditepe University, Turkey, 34855. E-mail:{sgener, sgoren}@cse.yeditepe.edu.tr*
- *L.-S. Didier is with Université du Sud Toulon Var, France. E-mail: Laurent-Stephane.Didier@univ-tln.fr*

Manuscript received October 28, 2016; revised xxxxx.



We first compute the Euclidean division of 7 by 3. This gives the first digit of the quotient, here 2, and the remainder is 1. We now have to divide 176 by 3. In the second iteration, we divide 17 by 3: the second quotient digit is 5, and the remainder is 2. The third iteration divides 26 by 3: the third quotient digit is 8, the remainder is 2, and this is also the remainder of the division of 776 by 3.

Fig. 1. Illustrative example: division by 3 in decimal

1.2 An introductory example

Let us introduce the proposed family of techniques with the help of usual decimal arithmetic. Suppose we want to divide an arbitrary number, say 776, by 3. Fig. 1 describes the paper-and-pencil algorithm in this case.

The key observation is that, in this example, the iteration body consists of the Euclidean division by 3 of a 2-digit decimal number. The first of these two digits is a remainder from previous iteration: its value is 0, 1, or 2, but no larger. We may therefore implement this iteration with a look-up table (LUT), which, for each value from 00 to 29, gives the quotient and remainder of its division by 3. This small LUT will allow us to divide numbers of arbitrary size by 3.

1.3 Previous work

Division by a constant in a hardware context has actually been studied quite extensively [2], [3], [4], [5], with good surveys in [4], [6], [7]. There are two main families of techniques: those based on additions/subtractions and those based on multiplication by the reciprocal. The table-based techique studied in this article was introduced in [8]. Prior to that, it had, to our knowledge, only been described in lecture notes [9] as an example of combinational circuit. It is in essence a straightforward adaptation of the paper-and-pencil division algorithm in the case of small divisors. The reason why this technique is not mentioned in the literature

is probably that the core of its iteration itself computes a (smaller) division: it does not reduce to either additions or multiplications. However, it is possible to express this smaller division as a small LUT.

Tabulating a complex function is sometimes an efficient technique, and it is the case here. In particular, the proposed technique is very well suited to modern FPGAs, whose reconfigurable logic resources are based on 4- to 6-input LUTs. Nonetheless, this technique is also evaluated in this article for application-specific integrated circuits (ASICs). It is well-known that the optimal architecture for addition or multiplication can be very different on these two targets, ASIC and FPGA. One contribution of this article is to show quantitatively that this is also true for table-based constant division methods.

The proposed architectures will be compared with the state-of-the-art of reciprocal-based architectures of [4], which builds upon [10]. This architecture is called **Recip** in the following. [4] shows how to determine three integers A, B and p such that

$$\left\lfloor \frac{X}{D} \right\rfloor = \left\lfloor \frac{AX+B}{2^p} \right\rfloor \quad \forall X \in \{0,\dots,2^{n-1}\} \tag{1}$$

More specifically, it first determines the minimal bitwidth w of A such that (1) is possible. There are two possible choices for w in [4], and the smallest one is chosen. In the present work, the value of w chosen is not always the smallest one, but the one that minimizes the area of the corresponding multiplier by A. Otherwise, our Recip reimplementation is faithful to [4], which indeed provides the minimal value of w.

1.4 Outline of the article

Section 2 adapts the radix-10 algorithm demonstrated on Fig. 1 to a radix that is a power of two. On LUT-based FPGAs, this radix is chosen so that the algorithm's LUTs match well with the FPGA's hardware LUTs. The linear architecture obtained by unrolling this recurrence was introduced in [8] and is called **LinArch** throughout this paper.

Section 3 studies variations of this recursion that lead to binary tree implementations with shorter latency. These architectures are called **BTCD** (for Binary Tree Constant Divider) and generalize those introduced in [11].

Section 4 considers the case when only the quotient or only the remainder are needed.

Section 5 studies the case when the divider is a product of two small numbers.

Finally, Section 6 studies the division by a small constant of a floating-point input. It shows that it is possible to ensure correct rounding to the nearest with very little overhead.

1.5 Methodology

All methods presented here have been implemented in two hardware generators. One is the open-source project FloPoCo[1]. The object of FloPoCo is application-specific arithmetic for FPGA computing, and this generator has been used to obtain most FPGA results. The other one is a generator of Verilog written in Perl at Ozyegin and Yeditepe Universities. Both generators are completely parameterized and can generate circuits for any dividend size and divisor value, and power-of two radix. Each generator also includes a test framework that has been used to validate the generated architectures.

1. http://flopoco.gforge.inria.fr/

TABLE 1
Notations used in this article

X, D, Q, R	dividend, divisor, quotient and remainder such that $X = DQ + R$ with $0 \leq R < D$
k	size of a chunk of X, i.e., X and Q are considered in radix 2^k
r	size in bits of $D-1$
X_i, Q_i, etc	sub-words (or: digits in radix 2^k) of X, Q, etc.
n	number of bits of the dividend X
m	number of radix-2^k digits of the dividend X
ℓ	nominal LUT input size in the target FPGAs

The FPGA synthesis results in this paper were obtained with Xilinx ISE 14.7, targetting a high-speed Virtex6 (part 6vhx380tff1923-3). These are synthesis results before place and route, which is perfectly meaningful for such small operators. The ASIC synthesis results have been obtained with Synopsys Design Compiler using the worst-case version of TSMC Artisan 180nm generic stdcell library with a wire-load model. The inputs and outputs are connected to flip-flops in a wrapper module, and the synthesis optimizes for the critical path.

2 Basic recurrence and linear architecture

Let D be the constant divisor, and let k be a small integer. We will use the representation of the input dividend X in radix 2^k, which may also be considered as breaking down the binary decomposition of X into m chunks of k bits (see Fig. 3):

$$X = \sum_{i=0}^{m-1} X_i.2^{ki} \quad \text{where} \quad X_i \in \{0,...,2^k-1\} \tag{2}$$

In this article, we assume that D is not a multiple of 2, as division by 2 reduces to a constant shift, which is simply wiring in a circuit handling binary data.

2.1 Algorithm

The following algorithm computes the quotient Q and remainder R of the high radix Euclidean division of X by constant D. In each step of this algorithm, the partial dividend, Y_i, the partial remainder, R_i, and one radix-2^k digit of the quotient, Q_i, are computed.

The line $Y_i \leftarrow X_i + 2^k R_{i+1}$ is simply the concatenation of a remainder and a radix-2^k digit. This corresponds to "dropping a digit of the dividend" in Fig. 1.

Let us define r as bitwidth of the largest possible remainder:

$$r = \lceil \log_2(D-1) \rceil \tag{3}$$

Note that r this is also the bitwidth of D, as D is not a power of two. Then, Y_i is of size $k+r$ bits. The second line of the loop body, $(Q_i, R_i) \leftarrow (\lfloor Y_i/D \rfloor,\ Y_i \mod D)$, computes a

Algorithm 1 LUT-based computation of X/D

procedure CONSTANTDIV(X, D)
  $R_m \leftarrow 0$
  for $i = m-1$ **down to** 0 **do**
    $Y_i \leftarrow X_i + 2^k R_{i+1}$
    (This + is a concatenation)
    $(Q_i, R_i) \leftarrow (\lfloor Y_i/D \rfloor,\ Y_i \mod D)$
    (read from a table)
  end for
  return $(Q = \sum_{i=0}^{m-1} Q_i.2^{ki}, \quad R = R_0)$
end procedure

radix-2^k digit and a remainder: it may be implemented as a LUT with $k+r$ bits of input and $k+r$ bits of output (Fig. 2).

Theorem 1. *Algorithm 1 computes the Euclidean division of X by D: It outputs the quotient Q and the remainder R so that $R = Q \times D + R$. The radix-2^k representation of the quotient Q is also a binary representation, each iteration producing k bits of this quotient.*

Proof. The proof proceeds in two steps. First, Lemma 1 states that $X = D \times \sum_{i=0}^{m} Q_i.2^{-ki} + R_0$. This shows that we compute some kind of Euclidean division, but it is not enough: we also need to show that the Q_i form a binary representation of the result. For this, it is enough to show that they are radix-2^k digits, which is established through Lemma 2.

Lemma 1.

$$X = D \sum_{i=0}^{m} Q_i.2^{-ki} + R_0$$

Proof. By definition of Q_i and R_i we have $Y_i = DQ_i + R_i$.

$$\begin{aligned} X &= \sum_{i=0}^{m-1} X_i.2^{-ki} \\ &= \sum_{i=0}^{m-1} (X_i + 2^k R_{i+1}).2^{-ki} \\ &\quad - \sum_{i=0}^{m-1} (2^k R_{i+1}).2^{-ki} \qquad \text{and} \\ &= \sum_{i=0}^{m-1} (DQ_i + R_i).2^{-ki} - \sum_{i=1}^{m} R_i.2^{-ki} \\ &= D \sum_{i=0}^{m-1} Q_i.2^{-ki} + R_0 - R_m.2^{-km} \end{aligned}$$

$R_m = 0$. □

Lemma 2. $\forall i \quad 0 \le Y_i \le 2^k D - 1$

Proof. The digit X_i verifies by definition $0 \le X_i \le 2^k - 1$; R_{i+1} is either 0 (initialization) or the remainder of a division by D, therefore $0 \le R_i \le D-1$. Therefore $Y_i = X_i + 2^k R_{i+1}$ verifies $0 \le Y_i \le 2^k - 1 + 2^k(D-1)$, or $0 \le y_i \le 2^k D - 1$. □

We deduce from the previous lemma and the definition of Q_i as quotient of Y_i by D that

$$\forall i \quad 0 \le Q_i \le 2^k - 1$$

which shows that the Q_i are indeed radix-2^k digits. Thanks to Lemma 1, they are the digits of the quotient. □

The algorithm computes k bits of the quotient in each iteration: the larger k is, the fewer iterations are needed for a given input number with bitwidth n. However, the larger k is, the larger the required LUT. Section 2.3 studies this trade-off, both for ASIC and FPGA.

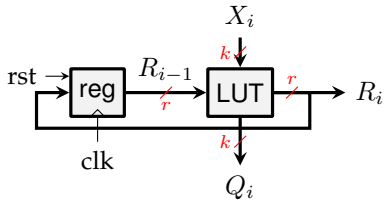


Fig. 2. Sequential architecture for Algorithm 1: LUT-based division of a number written in radix-2^k by a constant

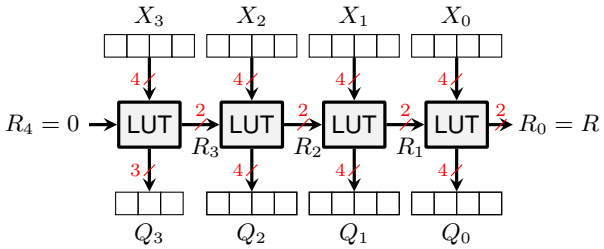


Fig. 3. Unrolled architecture (LinArch) for Algorithm 1: LUT-based division by 3 of a 16-bit number written in radix 2^4 ($k = 4$, $r = 2$)

2.2 Iterative or unrolled implementation of the basic recurrence

The iteration may be implemented sequentially as depicted in Fig. 2 or as the fully unrolled architecture depicted in Fig. 3. In all of the following, we will focus on the latter, which we denote by LinArch (short for linear architecture), because it enables high-throughput pipelined implementations.

2.3 Cost evaluation of LinArch

Algorithm 1 performs $\lceil n/k \rceil$ iterations. For a given D and k, the architecture therefore grows linearly with the input size n. Note that general division or multiplication architectures grow quadratically. Let us now consider how it grows with the constant D.

The area of a LUT in ASIC is essentially proportional to the number of bits it holds. One LUT of Fig. 2 and 3 needs to store $D \cdot 2^k$ entries of $r+k$ bits, or $D \cdot 2^k(r+k)$ bits. To remove D from this formula, we may round up the table size to a power of two: one table will store $2^{r+k}(r+k)$ bits, some of which are "don't care" values.

Finally, the area cost of the sequential architecture grows as $2^{r+k}(r+k)$, and that of LinArch as $\lceil n/k \rceil 2^{r+k}(r+k)$.

The delay of a LUT is essentially proportional to the number of input bits: the delays of both architectures grow as $\lceil n/k \rceil (r+k)$. Section 3 will introduce a parallel architecture that offers with smaller delay, sometimes at the expense of larger area.

2.4 The particular case of FPGAs

2.4.1 Using logic LUTs

The basic reconfigurable logic block of current FPGAs is a small LUT with 4 to 6 inputs and one output. We denote it LUTℓ in the following, with $\ell = 4$ to $\ell = 6$. In our case, these LUTℓ can also be used to build the $r+k$-input, $r+k$-output LUTs we need.

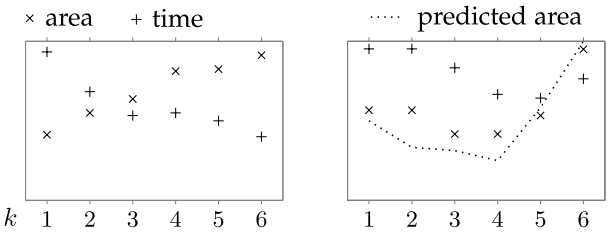


Fig. 4. Area and delay of LinArch in ASIC (left) and FPGA (right) as a function of k ($n = 16$, $D = 3$).

If the unit cost of FPGA logic is the LUTℓ, there is no point in using tables with fewer than ℓ inputs. We therefore choose k such that $r+k \geq \ell$. Then the cost of an $r+k$-input, $r+k$-output LUT is no longer $2^{r+k}(r+k)$ but $2^{r+k-\ell}(r+k)$.

Finally, the area of LinArch in LUTℓ is $\lceil n/k \rceil 2^{\max(r+k-\ell,0)}(r+k)$.

2.4.2 Choosing the optimal k for a given D and a given FPGA

As Fig. 4 illustrates, the optimal choice of k in terms of area is the smallest k such that $r+k \geq \ell$. As $r = \lceil \log_2(D-1) \rceil$, the method is very efficient for small values of D.

In each FPGA family, there are restrictions on LUT utilization. In Altera Stratix IV to 10, the Adaptive Logic Module (ALM) can be used as two arbitrary LUT4, but may also implement two LUT5 or two LUT6 under the condition that they share some of their inputs. For instance, a 6-input, 6-output LUT may be built as 3 ALMs.

In Xilinx series 5 to 7, the logic slice includes 4 registers and 4 LUT6s, each of which is fractionable as two LUT5 with independent outputs. The sweet spot here is therefore to build 5-input tables, unless we need to register all the outputs, in which case 6-input tables should be preferred.

We may use, for instance, 6-input LUTs to implement division by 3 ($r = 2$) in radix 16 ($k = 4$), as illustrated by Fig. 3. Implementing the core loop costs 6 LUTs (for a 6 bits in, 6 bits out table). The cost forthe complete LinArch for n bits is $\lceil n/4 \rceil \times 6$ LUT6s, for instance 36 LUT6s for 24 bits (single precision), or 78 LUTs for 53 bits (double precision).

Table 2 reports some synthesis results on Xilinx Virtex6 obtained using ISE 14.7. The purpose of this table is to show that the estimation formulae provide good predictions of the area. A general comparison with other division techniques will be provided in the following sections.

This table also shows that the method is mostly suited to small constants: for this FPGA, where $\ell = 6$, from $D = 17$, we have $k = 1$, so the architecture requires as many LUTs as there are bits in the input. Besides, the size of these LUTs then grow as the exponential term 2^{r+k-l}.

2.4.3 Using embedded memory blocks

For larger tables, a second option is the embedded memory block, from 9 Kb to 144 Kb depending on the architecture.

For division by 3, we may also use embedded memories with $k = 7$ to 11. More importantly, embedded memories push the relevance of this technique to larger constants.

These memories are not combinational, their inputs must be registered: they are best suited to either sequential or unrolled but pipelined implementation.

TABLE 2
Performance of various dividers of a 32-bit value by D (LinArch version) on Virtex-6

D	r	k	est. area	meas. area	meas. delay
3	2	4	48	47	5.5 ns
5	3	3	66	60	6.9 ns
7	3	3	66	60	6.9 ns
9	4	2	96	89	10.2 ns
17	5	1	192	193	19.0 ns

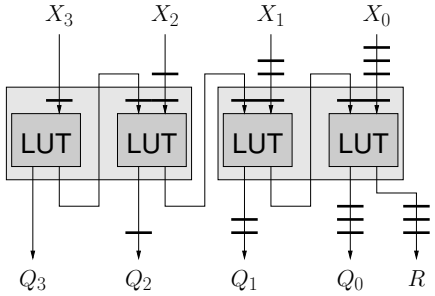


Fig. 5. A pipelined divider using two dual-ported embedded RAMs

In the latter case, we may exploit the fact that all these embedded memories are dual-ported: two iterations may be unrolled in one single memory block as depicted in Fig. 5. Again for division by 3, exploiting the M9K blocks of a Stratix IV (using $k = 7$), a fully pipelined single-precision divider by 3 could be impemented in 2 M9K only (2*2*7=28 bits) and run in 4 cycles at the maximal practical speed supported by these devices.

3 Parallel division

In this section, we present the family of *Binary Tree Constant Division* (BTCD) circuits. BTCD is to LinArch what fast adders are to carry-propagate adders: its latency has near logarithmic growth in n (versus the linear growth of LinArch), however its area is typically larger.

There are 6 fundamental variants of BTCD. On the one hand, a BTCD can be naive, timing-driven, or area-driven (i.e., 3 options). On the other hand, it may be based on regular adder or carry-save ones (i.e., 2 options). When these are paired, we get $3 \times 2 = 6$ options in total. The architecture proposed in [11] is "BTCD naive" with regular adders, and we call it simply BTCD. Timing-driven, area-driven, and carry-save are indicated with 't', 'a', and 'c' suffixes, respectively. Hence, the newly proposed versions are BTCDt, BTCDtc, BTCDa, BTCDac, and BTCDc.

3.1 The basics of BTCD

As for LinArch we first illustrate the method with an example in decimal (Fig. 6). BTCD first subdivides the dividend into m digits. On the example we can write this

$$X = X_8 = \sum_{i=0}^{7} X_{8,i} 10^i \tag{4}$$

Then each digit $X_{8,i}$ is divided by D in parallel, leading to m quotient digits and m remainder digits:

$$X_{8,i} = DQ_{8,i} + R_{8,i} \tag{5}$$

therefore (4) becomes

$$X_8 = D\sum_{i=0}^{7} Q_{8,i}10^i + \sum_{i=0}^{7} R_{8,i}10^i \tag{6}$$

On Figure 6, the $Q_{8,i}$ are written on the right-hand side, and can be considered a single decimal number $Q_8 = \sum_{i=0}^{7} Q_{8,i}10^i = 22065423$. The $R_{8,i}$ are written on the left-hand side, and can similarly be considered a single decimal number $R_8 = \sum_{i=0}^{7} R_{8,i}10^i = 11100000$, so (4) can be simply written

$$X_8 = D \times Q_8 + R_8 \tag{7}$$

Now let us define $X_4 = R_8$, and let us group its digits two by two: $X_4 = 22\ 06\ 54\ 23$. This is simply a rewriting in a radix-100 representation:

$$\begin{aligned} X_4 = R_8 &= \sum_{i=0}^{7} R_{8,i}10^i \\ &= \sum_{i=0}^{3} (10R_{8,2i+1} + R_{8,2i})100^i \\ &= \sum_{i=0}^{3} X_{4,i}100^i \end{aligned} \tag{8}$$

Since each radix-10 digit $R_{8,i}$ is a remainder of the division by 7, we have

$$0 \le X_{4,i} < 66. \tag{9}$$

The next step is to write the Euclidean division of X_4 by D:

$$\begin{aligned} X_4 &= \sum_{i=0}^{3} X_{4,i}100^i \\ &= D\sum_{i=0}^{3} Q_{4,i}100^i \ + \ \sum_{i=0}^{3} R_{4,i}100^i \\ &= DQ_4 \ + \ R_4 \end{aligned} \tag{10}$$

From (9), the Euclidean division by 7 of each $X_{4,i}$ digit will lead to a quotient Q_4 smaller than 10 (hence fitting on one decimal digit), and a remainder between 0 and 6 (also fitting one decimal digit). This explains the apparition of zeroes on both sides of Fig. 6, where $Q_4 = 03000703$ and $R_4 = 01060502$.

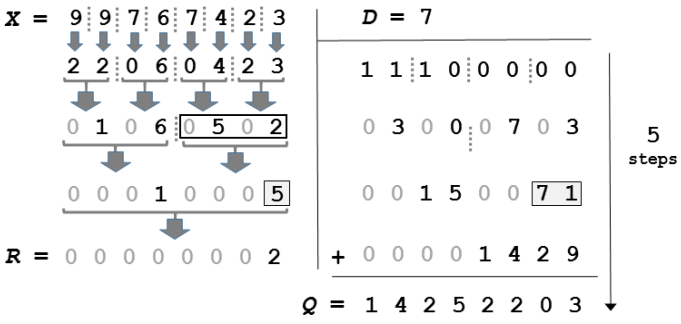


Fig. 6. BTCD's algorithm is shown with radix-10 and $D = 7$.

Finally we can again pair two radix-100 digits to obtain a radix-10000 representation of Q_4 and $R_4 = X_2$, and start again: (7) can be rewritten as

$$\begin{aligned} X_8 &= D \times Q_8 + X_4 \\ &= D \times Q_8 + (D \times Q_4 + X_2) \\ &= D \times Q_8 + D \times Q_4 + D \times Q_2 + D \times Q_0 + R_0 \\ &= D \times (Q_8 + Q_4 + Q_2 + Q_0) + R_0 \\ &= D \times Q + R \end{aligned} \tag{11}$$

As $0 \le R_0 < D$, we deduce that Q computed this way is indeed the quotient of the division of X by D.

3.2 Architecture

All this can be generalized it to a dividend of 2^m digits in radix-2^k.

$$\begin{aligned} X = X_{2^m} &= D \times \sum_{j=0}^{m} \sum_{i=0}^{2^j-1} Q_{2^j,i}(2^k)^{i2^{m-j}} + R_0 \\ \text{while} \quad X_{2^{j-1},i} &= R_{2^j,2i+1}(2^k)^{2^{m-j}} + R_{2^j,2i} \end{aligned} \tag{12}$$

Note that, at every stage in our recurrence, we pair remainder digits to form a single digit of the reduced dividend.

The main idea of BTCD is that each Euclidean division in this process can be performed in a LUT: although the radix is squared at each level, the digits manipulated have a constant number of non-zero digits (two decimal digits in our example). Every arrow in Fig. 6 corresponds to a LUT.

The first level of LUTs is different from the other levels. They are named iLUT for "initial LUT" (see Fig. 7), while the LUTs of other levels are named rLUT for "remainder LUT" (see Fig. 8).

The overall topology of a BTCD naive is a balanced binary tree and is given in Fig. 7, while the internals of a cBLK (the nodes of the second level and down) are given in Fig. 8.

We can also think of BTCD as two parallel binary trees. One binary tree is the binary tree of arrows on the left in Fig. 6, where every arrow is a LUT. The top row is iLUTs, and other arrows are rLUTs. Every LUT outputs a quotient (Q) and remainder (R). The Rs go from one LUT to another, while the Qs run to the right where the partial quotients are. On the right there is another binary tree—a binary tree of adders. That is, copy the binary tree of arrows on the left, excluding the row of smaller arrows at the top, and paste it on the right, so that the first row is between the two partial quotients of 11100000 and 03000703.

Consider the numbers 0502, 5, and 71, which are marked with boxes around them in Fig. 6. The arrow under 0502 is an rLUT that outputs an R of 5 and a Q of 71. There is an adder at the same position on the right (in the area of partial quotients) that takes the sum of 00 and 07 concatenated with the sum of 00 and 03 and adds it with 71. Note that the zeros that are pale in color are digits that happen to be zero at all times, no matter what the original dividend and divisor are.

The single binary tree topology shown in Fig. 8 is obtained by overlaying the left and right binary trees described above. In this tree, the top row of nodes is iLUTs, while the

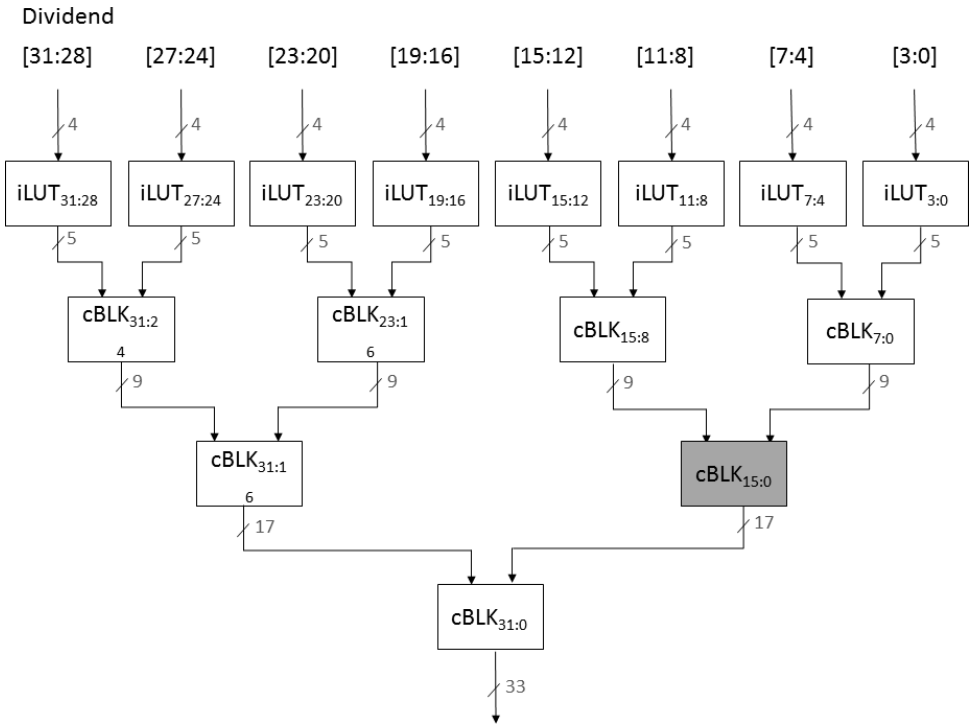


Fig. 7. BTCD circuit for the division in Fig. 6.

lower rows contain logic from both left and right binary trees. We call those lower nodes cBLK, short for combiner BLocK. A combiner block combines the left tree and right tree.

Consider again the numbers in boxes in Fig. 6 (0502, 5, and 71). The cBLK that corresponds to these numbers is responsible for hex-digit positions 3 through 0 (hence bit positions 15:0) and is shaded in Fig. 7. That is, for a dividend of 5423 this cBLK gives the final quotient of 0774 and remainder of 5. Its inputs come from the two cBLKs that are responsible for bit positions 15:8 and 7:0. Since the cBLKs providing inputs are responsible for disjoint digits, their quotient outputs can be concatenated. The same applies to the remainders.

The input and output bitwidths of cBLKs grow as we go down the binary tree from iLUTs. Internally, the number of words in rLUTs is the same in all cBLKs but their wordsize grows. The adder width grows as well.

In Fig. 7, the subscript of a submodule shows for which bit range of the dividend it does division. Similarly, in Fig. 8, the subscripts of Q and R show for which bit range of the dividend they are respectively the quotient and remainder. As shown in Fig. 7, cBLK15:0 receives a total of 9 bits of input from each of the parent LUTs, namely, cBLK15:8 and cBLK7:0. cBLK7:0 produces Q_R and R_R of Fig. 8, which are respectively the quotient and remainder that result from the bits [7:0] of the dividend. A 8-bit binary-coded decimal number divided by 7 can at most be 5 bits. On the other hand, remainder out of all subblocks can be at most 3 bits (maximum value of 6). The total number of bits that go from cBLK7:0 to cBLK15:0 is hence 9 bits (5+4).

An iLUT block in Fig. 7 has no internal structure unlike cBLK; it is simply a LUT. It has an input of k bits and has $2k$ words. Each word has a width that is equal to the number of

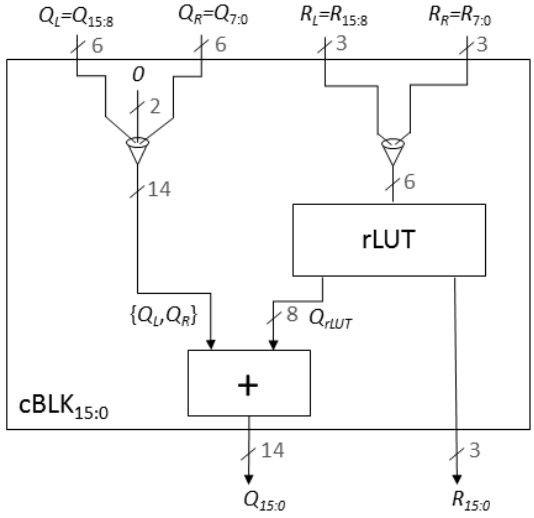


Fig. 8. Internals of cBLK, specifically cBLK15:0 of Fig. 7 (shaded).

output bits, namely, quotient and remainder bits. The quotient output is as many bits as the unsigned number $\frac{2k-1}{D}$, where D is the constant divisor. The remainder output, on the other hand, is as many bits as the unsigned number $D1$ (call it r bits). Assuming D is odd, the number of output bits of iLUT is k, and hence, the total number of bits of iLUT is $k2^k$.

An rLUT, on the other hand, has an input of $2r$ bits, actually representing a number with $k2^l + r$ bits, where l is the number of the level, level 0 being the level of iLUTs. The left r bits of input to the rLUT (R_L) come from the parent rLUT in the binary tree to the upper left of the rLUT we are speaking of, and right r bits (R_R) come from the parent

rLUT to the upper right. Therefore, the number represented by the two sets of r bits is equal to $(R_L << k2^l)+R_R$, which is a binary number that looks like xx..x00..0xx..x, where xs on the left are R_L, then there are $k-r$ zeros, and then xs on the right are R_R. Every rLUT contains D^2 words, which is relatively small. Since a LUT is a ROM (Read Only Memory), its input is an address (say m bits), and it has 2^m words. However, in the case of rLUT, since many bits of the input are zeros and since the non-zero bits can be at most $D-1$, we have much fewer words than 2^m words.

3.3 BTCD with optimized partitioner versus BTCD naive

BTCD was initially proposed in [11]. We have later came up with more sophisticated versions of BTCD, which will be explained in this subsection and the next. The original BTCD ([11]) is based on a rather bottom-up process, while the new versions are based on a top-down approach, where there is optimization involved in the divide-and-conquer process.

The original BTCD architecture (a.k.a. BTCD naive and abbrev. BTCD) divides the dividend into chunks of k bits (a design parameter) and solves these small instances of division by paralel iLUTs. Then, it combines them in pairs through rLUTs and adders thus solving larger and larger division problems (i.e., with more input bits). BTCD naive works with any input bitwidth although we report synthesis results with bitwidths that are powers of two. Note that we have tried chunk sizes from 1 to 6 and odd iLUT sizes and unpaired chunks may occur even with input bitwidths of power of two. When not all chunks pair, we bypass the output of the chunk on the right hand side down to the next level.

We propose in this article BTCD variants with two "optimized partitioner"s, one of which is "timing-driven", and the other is an "area-driven" partitioner. The ideas behind both are the same except the optimization criteria are different. The BTCD partitioner splits the n-bit input of constant division into n_{Left} and n_{Right} bits, for respectively left and right pieces, such that:

$$n = n_{Left} + n_{Right} \tag{13}$$

When BTCD partitioner breaks the size-n problem into two smaller and independent problem instances of size-n_{Left} and size-n_{Right}, the area or the timing of the size-n circuit can be formulated as $f(n)$:

$$f(n) = min(f_{iLUT}(n), min_{n_{Left}}\{g(n_{Left}, n_{Right})\}) \tag{14}$$

Above, $f(n)$ is a recursive function, which recurses indirectly through g. It computes the minimum area when $f = f_A$ and $g = g_A$, while it computes the minimum timing when $f = f_T$ and $g = g_T$. Therefore, the only difference in area-driven and timing-driven partitioners is in function g.

$$\begin{aligned} &g_A(n_{Right}, n_{Left}) = \\ &\quad g_{AcBLK}(n_{Left}, n_{Right}) + f_A(n_{Left}) + f_A(n_{Right}) \end{aligned} \tag{15}$$

Let us first explain the area-specific formulation shown in Eq. 15. The combined divider is composed of the left divider (n_{Left} bits of input), right divider (n_{Right} bits of input), and cBLK (combiner circuit). Hence, the total area equals the sum of the areas of the left divider ($f_A(n_{Left})$) and right divider ($f_A(n_{Right})$) plus the area of cBLK block shown in Fig. 8 ($g_{AcBLK}(n_{Left}, n_{Right})$, a non-recursive function).

$$\begin{aligned} &g_T(n_{Right}, n_{Left}) = \{g_T^R, g_T^Q\} \\ &f_T(n) = \{f_T^R, f_T^Q\} \end{aligned} \tag{16}$$

$$\begin{aligned} g_T^R(n_{Right}, n_{Left}) = &g_{TcBLK}^{rLUT}(n_{Left}, n_{Right}) \\ &+ max(f_T^R(n_{Left}), f_T^R(n_{Right}) \end{aligned} \tag{17}$$

$$\begin{aligned} &g_T^Q(n_{Right}, n_{Left}) = \\ &g_{TcBLK}^{Add}(n_{Left}, n_{Right}) + \\ &max(g_T^R(n_{Right}, n_{Left}), f_T^Q(n_{Left}), f_T^Q(n_{Right}))) \end{aligned} \tag{18}$$

On the other hand, the timing version of the formulation is more complicated, because timing values constitute a vector. At the output of cBLK, Q and R have different settling times, hence, g_T and f_T are both 2D vectors. Coincidentally, f_{cBLK} also has 2 components, namely, the delay of rLUT (g_{TcBLK}^{rLUT}) and the delay of adder (g_{TcBLK}^{Add}). As can be seen in Fig. 8, the timing of R output of cBLK, namely, g_T^R, can be computed by adding g_{TcBLK}^{rLUT} to the larger one of $f_T^R(n_{Left})$ and $f_T^R(n_{Right})$. That also gives us the timing of Q output of rLUT, which enters the adder. Therefore, the timing of the Q output of cBLK can be computed by finding the input arriving the latest (hence the max operation in Eq. 18).

Now, we will discuss the partitioning algorithm, which is summarized by Eq. 14. A size-n divider can be implemented $n+1$ different ways. It can be implemented directly by an iLUT (i.e., no partitioning) or n_{Left} can be any of $\{n-1,\ n-2,\ ..,\ 2,\ 1\}$ with corresponding n_{Right}s of $\{1,\ 2,\ ..,\ n-2,\ n-1\}$. Interestingly, the search set of possible partitions can be significantly narrowed down. Consider a partition of $(n_{Left} = a, n_{Right} = b)$ versus $(n_{Left} = b, n_{Right} = a)$, where $a > b$. The two circuits are identical except for the final cBLK as they both contain a size-a and size-b divider. As for the cBLK, the rLUTs contain the same number of entries (D^2) but their wordsizes are different ($b+r$ bits for (a,b) partition versus $a+r$ bits for (b,a)). Also, the cBLK of (a,b) adds an n-bit number with a b-bit number, while (b,a) adds an n-bit number with a a-bit number. Thus, the cBLK of (a,b) would be definitely smaller than (b,a) and slightly faster (due to the adder). Consequently, it is enough to consider partitions with $a \geqslant b$. This reduces the possibilities by around half at each stage of partitioning. Also, we do not consider iLUTs (i.e., no partitioning) when $n \geqslant 7$ as area starts to blow up although better timing may be obtained.

Eq. 14 indicates that for $f(n)$ to be the best (i.e., minimum) solution, we should consider the iLUT solution as well as all partitions, that is, all n_{Left} values ($n_{Right} = n - n_{Left}$. The curly braces in $min_{n_{Left}}\{\}$ denotes that the min operation is over a set as opposed to min() being over two numbers. The min operation in the case of timing requires more explanation. It selects the min of a number of 2D vectors. For that, every vector needs to be reduced to a single number. One component of the vector is the timing of

R and the other is of Q. Given the design of cBLK (Fig. 8), the timing of Q is always longer compared to R. Therefore, min selects n_{Left} based on which one offers the smallest Q timing.

The problem at hand is not an exponential recursive search. It is a "dynamic programming problem". Since the left divider depends on only n_{Left}, right divider on n_{Right}, cBLK depends on only n_{Left} and n_{Right}, we can independently optimize each of the three subblocks. For instance, using a non-optimal left divider will not allow us to design a further optimized right divider or a further optimized cBLK. Therefore, when we arrive at a subproblem of $f(a)$ and solve it optimally, we may save it and reuse it when we arrive at the same problem in the recursion tree again. Since we report results for up to $n = 128$, we produced the optimal trees (separately for area and timing) starting from $n = 1$ and going up to $n = 128$ with an inceremnt of 1. Larger n numbers use the solutions for the smaller n values. In this approach, for each n, we just compute top-level partition decision, as the lower-level partitioning decisions happen to be made in already solved smaller-size partitioning problems.

The partitioner is, in a way, an estimator, and the key to estimation is how we estimate the area and timing of iLUT, rLUT, and Adder. Below are the formulations of all six estimators.

$$A_{iLUT} = ((49/16) * 2^k - 3) * (1 + k) \quad (19)$$

$$A_{rLUT} = ((49/16) * D^2 - 3) * (1 + levelBitwidth/2) \quad (20)$$

$$A_{Add} = ((5/4) * ceil(log_2(bitwidth)) + 10.5) * bitwidth \quad (21)$$

$$T_{iLUT} = 4k - 4/3 \quad (22)$$

$$T_{rLUT} = 4r - 4/3 \quad (23)$$

$$T_{Add} = 2 * ceil(log_2(bitwidth)) + 4 \quad (24)$$

We model a LUT (iLUT or rLUT) as a binary tree of MUX2s (Eq. 19 and 20, first level of which is reduced to wires 75% of the time due to logic synthesis and is reduced to an inverter 25% of the time. There is a parallel lane for each bit of the word. The term $(49/16) * words - 3$ formulates each lane's area in terms of two-input gates. That has to be multiplied by wordsize ($(1 + k)$ and $(1 + levelBitwidth/2)$ respectively for iLUT and rLUT). In the case of timing (Eq. 22 and 23), the formula is $4 * addressBits - 4/3$, where one unit corresponds to the delay of a two-input gate. Note that when the address bits of a LUT is narrower than that of $D - 1$, the LUT reduces to wires, and the above formulas can be ignored.

When formulating the adder (Eq. 21 and 24), we assume that it is a prefix-graph based fast adder with Ladner-Fischer topology. We ignore the impact of fanout on area and timing. Area is supposed to be $Nlog(N)/2$ complexity and timing $log(N)$, where N should be replaced with the bitwidth of the adder.

3.4 BTCD with carry-save

We have observed that quite often the critical path of BTCD goes through the adders (chained together). Although all quotients to sum could be input to a traditional compression tree, we do a 3:2 compression at every level as quotients "arrive" one by one. Chain of regular adders are replaced by 3:2 Carry-Save Adder (CSAs). It is still a linear chain but each stage is a CSA with one logic level of parallel full-adder cells instead of a $log(N)$ complexity adder. There is still a regular final adder. CSAs are placed in to cBLKs with the exception of the first level of cBLKs (after the iLUTs). Second level of cBLKs inherit two sums from above and produce a new quotient to make it 3 quotients. Then, every level cBLKs reduces 3 quotients to 2 quotients.

Replacing cBLK's adder with a CSA can be applied to all three kinds of BTCDs, namely, BTCD naive (BTCD becomes BTCDc), BTCD area-driven partitioner (BTCDa becomes BTCDac), and BTCD timing-driven partitioner (BTCDt becomes BTCDtc).

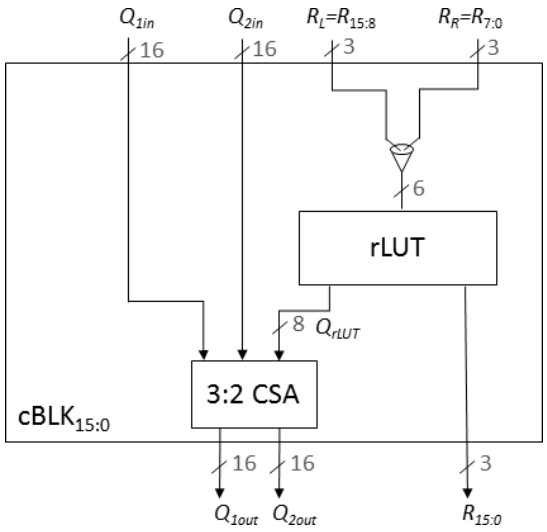


Fig. 9. Block cBLK with the 2-input Adder replaced by a 3:2 Carry-Save Adder.

3.5 Comparison of BTCD with LinArch and Reciprocal method

In order to identify the relevance domains of LinArch, BTCD, and Recip method, this section shows a generous amount of synthesis results of these three families of architectures, first for an ASIC standard-cell library as a target and then for Xilinx Virtex-6 FPGA.

3.5.1 ASIC synthesis results

We have generated 6 different BTCDs (hence the BTCD* in Table 3), LinArch, and Recip designs for divisors (D) of 3, 5, 11, and 23. And for each divisor, we have generated different versions of the design with a dividend bitwidth (n) of 8, 16, 32, 64, and 128. We also tried 3 different chunk sizes (k=3, 4, 5) for BTCD, BTCDc, and LinArch; BTCDt, BTCDtc, BTCDa, BTCDac, and Recip. do not have such parameter. We have a two-phase HDL generation approach on the ASIC side for BTCD*. Through the partitioners or the naive methods or

TABLE 3
Comparison of BTCD*, Reciprocal Method, and LinArch on ASIC

		Delay (ns)			Area/1000			ATP/1000		
d	n	LinArch	BTCD*	Recip	LinArch	BTCD*	Recip	LinArch	BTCD*	Recip
3	8	**1.9 (5)**	2.2 (4)	2.4	**2.7 (3)**	3 (a)	3.8	**5.4 (3)**	7.5 (a)	9.12
	16	**2.8 (5)**	3.3 (5)	3.5	**4.7 (3)**	8 (c5)	16	**16.8 (3)**	27.8 (5)	56
	32	4.9 (5)	**4.5 (ac)**	**4.5**	**10.3 (3)**	21.1 (c5)	60.7	**61.9 (3)**	97.06 (c5)	273.15
	64	8.6 (5)	**5.3 (tc)**	5.5	**19 (3)**	48.9 (c5)	209.3	**204.2 (3)**	264.06 (c5)	1151.15
	128	28.8 (4)	**5.8 (tc)**	6.5	**61.5 (4)**	101 (ac)	906.3	1772.6 (4)	**664.1 (tc)**	5890.95
complexity		n	$(\log n)^{1.1}$	$(\log n)^{1.2}$	$n^{1.1}$	$n^{1.3}$	n^2	$n^{2.1}$	$n^{1.6}$	$n^{2.3}$
5	8	**2 (4)**	2.1 (c3)	2.5	4.4 (3)	**4 (c5)**	4.6	8.9 (3)	**8.61 (c3)**	11.5
	16	3.8 (4)	**3.5 (5)**	**3.5**	10.4 (3)	**10.2 (c5)**	20.7	43.2 (3)	**38.76 (c5)**	72.45
	32	7.1 (4)	4.9 (c5)	**4.4**	**24.5 (3)**	26 (ac)	60.2	196 (3)	**143.08 (tc)**	264.88
	64	14 (4)	5.9 (c4)	**5.5**	**49 (3)**	56.7 (ac)	228.2	804.9 (3)	**394.32 (tc)**	1255.1
	128	28.4 (4)	**6 (c5)**	6.5	124.9 (4)	**101.4 (c5)**	918.1	3545.9 (4)	**608.4 (c5)**	5967.65
complexity		n	$(\log n)^{1.2}$	$(\log n)^{1.1}$	$n^{1.2}$	$n^{1.2}$	$n^{1.9}$	$n^{2.2}$	$n^{1.5}$	$n^{2.3}$
11	8	**1.7 (5)**	2 (c3)	2.5	7.4 (3)	**4.6 (ac)**	4.8	16.1 (3)	**11.2 (c3)**	12
	16	4.2 (5)	3.9 (4)	**3.5**	21.9 (3)	**14.2 (ac)**	17	106.7 (3)	76.9 (5)	**59.5**
	32	8.3 (5)	5.4 (5)	**4.3**	52 (3)	**33.5 (a)**	56.3	522.8 (3)	302.1 (5)	**242.09**
	64	17.5 (5)	7.1 (4)	**5.2**	106.9 (3)	**79.9 (ac)**	202	2222.3 (3)	**902.18 (tc)**	1050.4
	128	37.8 (4)	8.6 (4)	**6.3**	320.2 (4)	**168.5 (ac)**	757.7	12105.4 (4)	**2229.5 (c5)**	4773.51
complexity		$n^{1.1}$	$(\log n)^{1.7}$	$(\log n)^{1.1}$	$n^{1.4}$	$n^{1.3}$	$n^{1.8}$	$n^{2.4}$	$n^{1.9}$	$n^{2.2}$
23	8	**1.7 (5)**	**1.7 (4)**	3	8.7 (3)	5.6 (c3)	**4.1**	16.8 (4)	**11.76 (c3)**	12.3
	16	4.1 (5)	4 (4)	**3.3**	30.5 (3)	18.9 (a)	**17.2**	146 (3)	136.08 (a)	**56.76**
	32	8.5 (5)	5.9 (4)	**4.1**	78.3 (3)	**47.2 (a)**	50	845.5 (3)	573.93 (tc)	**205**
	64	17.1 (5)	7.7 (4)	**5.1**	169.1 (3)	**109.4 (a)**	186.6	3742.9 (3)	1815.46 (tc)	**951.66**
	128	43.2 (4)	9.3 (4)	**6.2**	511.8 (4)	**226.6 (ac)**	664.7	22103.4 (4)	4613.12 (tc)	**4121.14**
complexity		$n^{1.2}$	$(\log n)^2$	$(\log n)^{0.9}$	$n^{1.5}$	$n^{1.3}$	$n^{1.8}$	$n^{2.6}$	$n^{2.2}$	$n^{2.1}$

even manually we can specify a tree textually and input it to the HDL generator.

Our synthesis script does a binary search for the smallest latency with up to 4 synthesis runs for a given circuit. That is, we set a latency constraint and see if we meet it; if not, we raise it, if yes, we lower it.

Table 3 summarizes the delay, area, and ATP results we have obtained in a total of 1,120 synthesis runs we have performed.

In Table 3, the best (smallest) value in every category is marked in bold. Table 3 also displays the complexity trend of every type of result for every divisor value in terms of n. These are experimental formulas, and they more or less agree with the theory. In LinArch column of Table 3, the numbers in parentheses indicate the k value (chunk size) that gave the best result. In BTCD* column, however, there is a number in parenthesis if the BTCD with the best result is BTCD naive (hence BTCD or BTCDc). That is because only BTCD and BTCDc have a k parameter (tried 3, 4, 5). If there is a character in a cell in BTCD* column, it indicates the type of BTCD that gave the best result. No character in parenthesis means BTCD naive with regular adders won, which is the only and original BTCD suggested in [11].

Based on which method wins how many times, BTCD* are the best performer in area and ATP categories, while LinArch is the runner-up in area category.

Area results are critical in cases where pipelining and hence multiple-cycle latency is allowed. In Table 3, we have very competitive area results; even after adding the cost of pipeline flops to them, they are still quite comptetitive. One interesting outcome of the synthesis runs for Table 3 is that BTCDa and BTCDac ofer the smallest area in 55% of the time, more so than LinArch, which wins 35% of the time. Recip. ([4]) wins the remaining 10% of the cases, which are with 8 and 16-bit runs of $D = 23$.

ATP results have significance as they correlate with energy consumption. However, they can also be considered as a more universal area metric for the cases when the circuit is synthesized under more relaxed timing constraints. In ATP category, BTCD* win 50% of the time, while Recip wins 30%, and LinArch wins 20%.

In delay (a.k.a. latency or timing) category, the ranking is as follows Recip (55%), BTCD* (22.5%), LinArch (22.5%).

In terms of individual cases, we can state that LinArch wins with small D and small n in delay. It almost always wins with small D in area and ATP irrespective of n.

Recip is most competitive in delay and especially with large D and larger n. It is not much competitive in area. However, in ATP it is comptetive with large D and large n.

BTCD* is most competitive in area. It is almost equally competitive in ATP. It is more competitive with medium-size D.

3.5.2 FPGA synthesis results

Table 3.5.1 shows some synthesis results on Xilinx Virtex-6 obtained thanks to the FloPoCo implementation of the BTCD and reciprocal methods. In the reciprocal method, the constant multipliers of FloPoCo are used [12]. Instead of chosing the smallest constant size as in [4], this implementation choses the constant that leads to the smallest architecture (smallest number of *full-adder* cells).

4 Specialized Operators

This section discusses some variants of the previous architectures that are more efficient by only outputting the remainder or only outputting the quotient.

4.1 Reciprocal method outputting only the quotient

The reciprocal method as published in [4] only computes the quotient. The previous tables report results for architectures

TABLE 4
Comparison of LinArch, BTCD, and Reciprocal Method on Virtex-6 FPGA

		Delay(ns)			Area (LUT)		
D	wIn	LinArch	BTCD	Recip	LinArch	BTCD	Recip
3	8	**1.5**	2.0	3.1	**11**	19	35
	16	**2.8**	**2.8**	4.7	**23**	49	91
	32	5.5	**4.5**	5.7	**47**	115	218
	64	10.9	**6.0**	7.0	**95**	279	505
	128	21.6	**8.0**	9.7	**191**	646	1144
5	8	**1.6**	2.7	3.7	**12**	22	27
	16	3.5	**3.4**	4.7	**29**	57	89
	32	7.0	**4.5**	5.8	**60**	127	215
	64	14.2	**5.7**	7.1	**125**	325	501
	128	28.4	**7.4**	9.6	**252**	760	1139
11	8	**2.2**	2.7	4.6	**17**	27	48
	16	4.9	**4.3**	4.8	**41**	97	115
	32	10.2	**5.4**	5.8	**89**	205	224
	64	21.0	**6.9**	7.1	**185**	527	507
	128	42.4	**8.6**	9.7	**377**	1138+2BR	1403
23	8	3.0	**2.4**	5.4	**24**	26	44
	16	7.7	**4.0**	5.6	**67**	156	109
	32	19.1	**4.9**	7.3	**166**	306	251
	64	40.0	**6.1**	8.6	**355**	778 +5BR	570
	128	83.4	**7.8**	10.4	**744**	1649+10BR	1275

TABLE 5
The overhead of computing the remainder in Recip method. Recip/R denotes the architecture of [4] that computes quotient only.

		Delay(ns)		Area (LUT)	
D	wIn	Recip	Recip/R	Recip	Recip/R
3	8	3.1	3.1	35	33
	16	4.7	3.8	91	88
	32	5.7	5.1	218	215
	64	7.0	6.9	505	502
	128	9.7	9.6	1144	1141
23	8	3.4	3.4	44	27
	16	5.6	3.1	109	92
	32	7.3	5.1	251	234
	64	8.6	6.8	570	553
	128	10.4	9.6	1275	1258

that are slightly more complex, as they also compute the remainder as $R = X - DQ$. However, the overhead of this computation is very small, as Table 4.1 shows. Indeed, this computation has to be only performed on the number of bits of R, which is small with respect to the number of bits of X.

4.2 Remainder-only variant of LinArch and BTCD

In LinArch, if one only needs the remainder, the quotient bits need not be stored at all. This entails savings in terms of area that are easy to predict (roughly a factor $r/(r+k)$, as illustrated by Table 6). However, there is almost no improvement in delay, as the critical path is unchanged (see Fig. 3).

In BTCD, computing only the remainder improves both area and delay: if the quotient is not needed, there is no need for the corresponding part of the table nor the addition tree. All that remains is a binary tree of tables that have $2r$ inputs and r outputs. Note that there is a binary tree in [7] for this case, but it still involves additions.

Besides, as the critical path was in the quotient addition tree, the delay of a remainder-only architecture is significantly smaller than the delay of the complete architecture.

As Table 6 shows, BTCD therefore offers a much better area-time trade-off than LinArch if only the remainder is needed. For instance, for $k = 5$, the area is nearly identical (both architectures build the same number of LUTs with 6 inputs and 3 outputs) while the delay of BTCD is logarithmic instead of linear for LinArch.

5 Composite division

The table-based methods scale poorly with the size (in bits) of the constant. For a large constant that is the product of two smaller ones, it is however possible to divide successively by the two smaller constants. This will often lead to a smaller architecture than a monolithic division by the large constant. The following first formalizes this intuition, then discusses the choice of an optimal factorization.

5.1 Algorithm

Let us first assume that D is the product of two smaller constants:

$$D = D_a \times D_b \tag{25}$$

The Euclidean division of X by D_a can be written

$$X = D_a Q_a + R_a \quad \text{with } R_a \leq D_a - 1. \tag{26}$$

Then we can divide Q_a by D_b:

$$Q_a = D_b Q + R_b \quad \text{with } R_b \leq D_b - 1. \tag{27}$$

Putting all together, we obtain

$$\begin{aligned} X &= D_a D_b Q + D_a R_b + R_a \\ &= D \times Q + R \quad \text{where } R = D_a R_b + R_a. \end{aligned} \tag{28}$$

Since $R = D_a R_b + R_a \leq D_a(D_b - 1) + D_a - 1 = D - 1$, R is indeed the remainder of the division of X by D, and Q_b is indeed the proper quotient.

Now if the divisor D is the product of more than two factors (i.e. if D_a or D_b can themselves be factored), the previous decomposition can be applied recursively.

TABLE 6
Comparison of remainder-only architectures (R) to Euclidean divider architectures (Q+R)

D	wIn	Delay(ns)				Area (LUT)			
		LinArch		BTCD		LinArch		BTCD	
		Q+R	R	Q+R	R	Q+R	R	Q+R	R
3	8	1.5	1.5	2.0	1.5	11	4	19	6
	16	2.8	2.8	2.8	1.6	23	8	49	8
	32	5.5	5.4	4.5	2.8	47	16	115	24
	64	10.9	10.6	6.0	3.4	95	32	279	47
	128	21.6	21.0	8.0	4.1	191	64	646	96
5	8	1.6	1.6	2.7	1.6	12	6	22	6
	16	3.5	3.5	3.4	2.3	29	15	57	15
	32	7.0	6.9	4.5	2.9	60	30	127	33
	64	14.2	14.0	5.7	3.6	125	63	325	63
	128	28.4	27.9	7.4	4.3	252	126	760	129

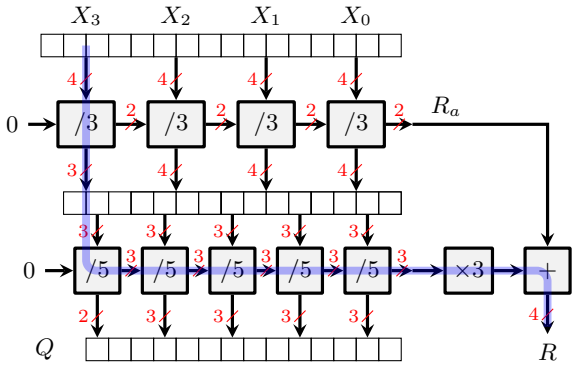


Fig. 10. Composite divider of a 16-bit value by $15 = 3 \times 5$, with the critical path highlighted

TABLE 7
Examples of composite 32-bit dividers on Virtex6. Estimates ignore the cost of the remainder reconstruction, which is accounted for in the measured area and delay. In all cases the optimal value of k is used.

D	decomp.	est. area	(by level)	meas. area	meas. delay
9	atomic	96	–	**89**	10.2 ns
	3×3	96	48+48	96	**6.7 ns**
15	atomic	96	–	**89**	10.2 ns
	5×3	114	(66+48)	109	8.3 ns
	3×5	114	(68+46)	110	**8.2 ns**
45	atomic	448	–	317	28.5 ns
	9×5	156	(96+60)	150	12.1 ns
	5×9	156	(66+90)	150	12.1 ns
	$5 \times 3 \times 3$	162	(66+48+48)	156	9.1 ns
	$3 \times 3 \times 5$	156	(48+48+60)	159	**8.9 ns**
	15×3	144	(96+48)	**136**	11.5 ns
	3×15	144	(48+96)	**136**	11.5 ns

TABLE 8
Examples of composite 32-bit dividers by 9 on ASIC.

D	decomp.	area / 1000	delay
9	atomic (LinArch)	29.8	8.5 ns
	atomic (BTCD)	47.3	**5.7** ns
	3×3 (LinArch, k=3)	**26.2**	8.6 ns
	3×3 (LinArch, k=5)	30.7	7 ns

5.2 Architecture

In terms of architecture, we need two of the previous dividers to compute the Euclidean divisions by D_a and D_b, plus a multiply-and-add to compute $r = D_a R_b + R_a$.

This additional multiply-and-add is typically small for two reasons. First, the remainders are small. Secondly, this is a constant multiplication, for which a range of techniques exist, some table-based [13], [14], some based on shift-and-add [12], [15], [16].

As can be seen on Fig. 10, the critical path is that of the divider with the largest m, plus one LUT for the quotient output, and a small constant multiplication and addition for the remainder output.

5.3 Results and discussions

Table 7 provides synthesis results on FPGAs for three constants that are of practical significance: 9 appears in some 2D stencils, 15 is (up to a power of two) 60 and occurs in timing conversions, and 45 is (up to a power of two) 360 which appears in degree/radian conversions. Interestingly, the two latter constants were probably chosen in ancient times because they can be divided by 2, 3, 4, and 5.

Table 7 shows that composite dividers are not only smaller, they are also consistently faster on FPGAs. This is easily explained on the example of division by 9. The atomic divider by 9 requires about the same area as the composite divider. However, its optimal value of k is 2, and the 32-bit input X is therefore decomposed in $m = 16$ radix-2^2 digits: the architecture has 16 LUTs on the critical path. When we compose two dividers by 3, optimal value of k is 4 for each, so the architecture of each sub-divider has only $m = 8$ LUTs on the critical path. As Fig. 10 shows, the critical path of the composite architecture is therefore $8 + 1$ LUTs only (plus the remainder reconstruction) instead of 16 LUTs.

The benefit of composite dividers on ASIC is much less spectacular. This very different behavior of ASIC and FPGA is a consequence of the very different curves shown by Fig. 4. Table 8 shows how composite dividers may marginally improve LinArch area (which was the optimal), but at the expense of delay. The best delay is still achieved by BTCD, but at a higher area. For larger dividends, the improvement is even more marginal, and there is a degradation in both area and delay as soon as the optimal atomic implementation is BTCD or Recip.

6 Floating-point division by a small integer constant

When accelerating floating-point computations on FPGAs, it makes sense to use operators that are tailored to the context. This section shows how to build a divider of a floating-point input X by a small integer constant that ensures bit-for-bit compatibility with a IEEE 754-compliant divider. A compiler of floating-point code to FPGA can use this operator as a drop-in replacement for a classical divider, at a fraction of the cost.

The proposed floating-point divider requires a Euclidean divider and can use for this any of the variants of the architectures previously presented (linear, composite, and/or parallel). For simplicity, the results are presented with LinArch.

A floating-point input X is given by its mantissa M and exponent E:

$$X = 2^E X_f \quad \text{with } X_f \in [1,2) \quad . \tag{29}$$

Similarly, the floating-point representation of our integer divisor D is:

$$D = 2^S D_f \quad \text{with } D_f \in [1,2) \quad . \tag{30}$$

One may remark that $S = r - 1$ if D is not a power of two.

The main issues to address are the normalization and rounding of the floating-point division of X by D according to the IEEE 754 standard [17].

6.1 Normalization

Let us write the division

$$\frac{X}{D} = \frac{X_f.2^E}{D} = \frac{2^S X_f}{D} 2^{E-S} = \frac{X_f}{D_f} 2^{E-S}. \tag{31}$$

As $\frac{X_f}{D_f} \in [0.5, 2)$, this is almost the normalized mantissa of the floating-point representation of the result:

- if $X_f \geq D_f$, then $\frac{X_f}{D_f} \in [1,2)$, the mantissa is correctly normalized and the floating-point number to be returned is
$$Y = \circ\left(\frac{2^S X_f}{D}\right) 2^{E-S} \tag{32}$$
where $\circ(z)$ denotes the IEEE-standard rounding to nearest even of a real z.
- if $X_f < D_f$, then $\frac{X_f}{D_f} \in [0.5, 1)$, the mantissa has to be shifted left by one. Thus, the floating-point number to be returned is
$$Y = \circ\left(\frac{2^{S+1} X_f}{D}\right) 2^{E-S-1} \quad . \tag{33}$$

The comparison between X_f and D_f is extremely cheap as long as D is a small integer, because in this case D_f has only r non-zero bits. Thus, the comparison is reduced to the comparison of these r bits to the leading r bits of X_f. As both X_f and D_f have a leading one, we need a comparator on $r-1$ bits. On FPGAs, this is a very small delay using fast-carry propagation.

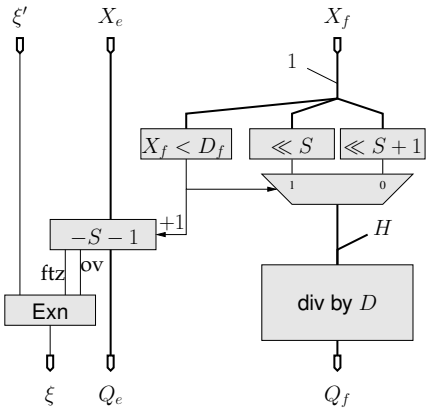


Fig. 11. Floating-point division by a small constant.

6.2 Rounding

Let us now address the issue of correctly rounding the mantissa fraction. If we ignore the remainder, the obtained result is the rounding towards zero of the floating-point division.

To obtain correct rounding to the nearest, a first idea is to consider the final remainder. If it is larger than $D/2$, we should round up, *i.e.* increment the mantissa. The comparison to $D/2$ would cost nothing (actually the last table would hold the result of this comparison instead of the remainder value), but this would mean an addition of the full mantissa size, which would consume some logic and have a latency comparable to the division itself, due to carry propagation.

A better idea is to use the identity $\circ(z) = \lfloor z + \frac{1}{2} \rfloor$, which in our case becomes

$$\circ\left(\frac{2^{S+\epsilon} X_f}{D}\right) = \left\lfloor \frac{2^{S+\epsilon} X_f + D/2}{D} \right\rfloor \tag{34}$$

with ϵ being 0 if $X_f \geq D_f$, and 1 otherwise. In the floating-point context we may assume that D is odd, since powers of two are managed as exponents. Let us write $D = 2H + 1$. We obtain

$$\begin{aligned} \circ\left(\frac{2^{S+\epsilon} X_f}{D}\right) &= \left\lfloor \frac{2^{S+\epsilon} X_f + H}{D} + \frac{1}{2D} \right\rfloor \\ &= \left\lfloor \frac{2^{S+\epsilon} X_f + H}{D} \right\rfloor \end{aligned} \tag{35}$$

so instead of adding a round bit to the result, we may add H to the dividend before its input into the integer divisor. It seems we haven't won much, but this pre-addition is actually for free: the addend $H = \frac{D-1}{2}$ is an S-bit number, and we have to add it to the mantissa of X that is shifted left by $S+\epsilon$ bits, so it is a mere concatenation. Thus, we save the adder area and the carry propagation latency.

To sum up, the management of a floating-point input adds to the area and latency of the mantissa divider those of one (small) exponent adder, and of one (large) mantissa multiplexer, as illustrated by Figure 11. On this figure, ξ is a 2-bit exception vector used to represent 0, $\pm\infty$ and NaN (Not a Number).

The implementation in FloPoCo manages such divisions by small integer constants and all their powers of two. The

TABLE 9
Floating-point division by a constant on Virtex-6

		single precision		double precision	
D	method	delay	area	delay	area
3	LinArch	5.2 ns	**70**	9.9 ns	198
	(mantissa only)	(4.7 ns)	(39)	(9.5 ns)	(82)
	BTCD	**4.9 ns**	117	**6.5 ns**	278
	Recip	5.5 ns	226	7.2 ns	554
	[3]	6.12 ns	156	7.46 ns	240
5	LinArch	6.5 ns	**84**	12.7 ns	172
	(mantissa only)	(6.15 ns)	(52)	(12.34 ns)	(108)
	BTCD	**5.0 ns**	133	**6.2 ns**	326
	Recip	5.2 ns	227	6.9 ns	547
	[3]	6.1 ns	155	8.43 ns	390

only additional issues are in the overflow/underflow logic (the Exn box on Figure 11), but they are too straightforward to be detailed here.

6.3 Results

Here, we only report FPGA results, because FPGA acceleration of floating-point code is probably the only context where floating-point constant dividers are useful.

In Table 9, the lines (mantissa only) illustrate the overhead of floating-point management. As expected, the area overhead is large but the timing overhead is very small, and both are almost independent of D.

This table also compares with the state-of-the-art, and there is another work to mention here: [3] is essentially a method that multiplies by the reciprocal using a shift-and-add tree that exploits the periodic binary representation of $1/D$.

7 Conclusion

This article adds division by a small integer constant (such as 3 or 10) to the bestiary of arithmetic operators available to C-to-hardware compilers. This operation can be implemented very efficiently, be it for integer inputs or for floating-point inputs.

The article studies qualitatively the performance (area and delay) of three families of techniques (linear, binary tree and multiplicative). On ASIC, each method has its relevance domain. On FPGAs, the simplest table-based methods behave comparatively better, thanks to the LUT-based hardware structure of FPGAs. Due to increasing routing pressure as technology progresses, the number of inputs to these FPGA LUTs keeps increasing. This should make this technique increasingly relevant in the future.

References

[1] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2003.

[2] E. Artzy, J. A. Hinds, and H. J. Saal, "A fast division technique for constant divisors," *Communications of the ACM*, vol. 19, pp. 98–101, Feb. 1976.

[3] F. de Dinechin, "Multiplication by rational constants," *IEEE Transactions on Circuits and Systems, II*, vol. 52, no. 2, pp. 98–102, Feb. 2012.

[4] T. Drane, W. Cheung, and G. Constantinides, "Correctly rounded constant integer division via multiply-add," in *Proc. Int. Symp. Circuits and Systems (ISCAS'2012)*, May 2012, pp. 1243–1246.

[5] S.-Y. R. Li, "Fast constant division routines," *IEEE Transactions on Computers*, vol. C-34, no. 9, pp. 866–869, Sep. 1985.

[6] P. Srinivasan and F. Petry, "Constant-division algorithms," *IEE Proc. Computers and Digital Techniques*, vol. 141, no. 6, pp. 334–340, Nov. 1994.

[7] R. W. Doran, "Special cases of division," *Journal of Universal Computer Science*, vol. 1, no. 3, pp. 67–82, 1995.

[8] F. de Dinechin and L.-S. Didier, "Table-based division by small integer constants," in *Applied Reconfigurable Computing*, Mar. 2012, pp. 53–63.

[9] A. Paplinski, *CSE2306/1308 Digital Logic Lecture Note, Lecture 8*, Clayton School of Information Technology Monash University, Australia, 2006.

[10] A. D. Robison, "N-bit unsigned division via n-bit multiply-add," in *International Symposium on Computer Arithmetic (ARITH)*, 2005, pp. 131–139.

[11] H. Ugurdag, A. Bayram, V. Gonul, and S. Gören, "Efficient combinational circuits for division by small integer constants," in *International Symposium on Computer Arithmetic (ARITH)*, Jul. 2016, pp. 1–7.

[12] N. Brisebarre, F. de Dinechin, and J.-M. Muller, "Integer and floating-point constant multipliers for FPGAs," in *Application-specific Systems, Architectures and Processors*. IEEE, 2008, pp. 239–244.

[13] K. Chapman, "Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner)," *EDN magazine*, no. 10, p. 80, May 1993.

[14] M. Wirthlin, "Constant coefficient multiplication using look-up tables," *Journal of VLSI Signal Processing*, vol. 36, no. 1, pp. 7–15, 2004.

[15] O. Gustafsson, "Lower bounds for constant multiplication problems," *IEEE Transactions On Circuits And Systems II: Express Briefs*, vol. 54, no. 11, pp. 974 – 978, Nov. 2007.

[16] M. Kumm, "Multiple constant multiplication optimizations for field programmable gate arrays," Ph.D. dissertation, Kassel Univ., 2016.

[17] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhauser Boston, 2009.