Scalable architecture of constant division on FPGA

Danila Gorodecky
INESC-ID, Instituto Superior Tecnico,
Universidade de Lisboa,
Portugal;
EHU / EPAM School of Digital Engineering,
Lithuania
Email: danila.gorodecky@gmail.com

Leonel Sousa
INESC-ID, Instituto Superior Tecnico,
Universidade de Lisboa,
Portugal
Email: leonel.sousa@tecnico.ulisboa.pt

***Abstract*—This paper proposes a method for hardware integer division by a constant, based only on combinational logic, i.e. without requiring storage and feedback in calculations. The proposed scheme for division consists of adders and encoders, where encoders are systems of Boolean functions. The proposed divisor provides at the output the quotient and the residue (at the same time or separately). Experiments conducted on FPGA demonstrate up to three times improvement in area cost compare to the optimized divisor circuits provided by the Xilinx tools, while the delay is improved by 25% for dividends with less than 48-bit. It is also shown in this paper that the proposed approach is scalable, and in comparison to the state-of-the-art, the proposed approach improves the area or the delay, or both for many constant values and input bit sizes.**

I. Introduction an state-of-the-art

The division is the most complex of the four fundamental arithmetic operations and, in general, does not produce an exact result. In modern microprocessors, integer division takes many clock cycles, comparable to floating-point division [1]. The target circuit area impacts how the result of the division can be generated: quotient, quotient and residue, approximation, precision of calculation [2], [3]. Overall, the implementation of a hardware division on FPGA and ASIC can pose some unique challenges, including the complexity of design, the high cost of development, the need for extensive verification and testing, and the need to protect intellectual property [2]–[7].

Based on functionality, implementation, performance, hardware architecture, etc., the division algorithms can be distinguished as iterative, recursive, variable latency, parallel, pipelined, sequential, slow, fast, and subtractive [2], [3], [6]–[18]. Non-restoring and restoring algorithms are two common approaches for performing hardware division in contemporary arithmetic units on FPGAs and ASICs [2], [3], [8], [10], [12], [15], [19]–[22].

Restoring-based and non-restoring-based division algorithm uses a sequence of subtractions and multiplications to compute the quotient and remainder. These algorithms work by repeatedly subtracting the divisor from the dividend and shifting the quotient and the remainder until the remainder is less than the divisor. Due to the specifics of the division, its hardware implementation typically requires memory, for instance, to store partial reminders, shift intermediate quotient, or control loop addition and subtraction, to perform the required set of iterations [23]–[27]. Electronic design automation tools generate hardware dividers for arbitrary values, for a limited bit-range of dividends and divisors. However, typically, they do not provide simultaneously the quotient and the remainder. For example, Xilinx 2015 and Synopsys 2013 and earlier versions can not generate general schemes of division/modulo calculation, only for very specific divisors such as $2^k - 1$.

The division by integer constants is an operation that often occurs in different areas and applications [28], such as counters [26], cryptography [29], [30], networks [24], microprocessors [31], [32], residue number system [33], and for accessing interleaved memory banks in numbers that are not powers of two. Hardware division by small integer constants was analysed in depth in [37]. The proposed recurrent and linear architectures, originally reported in [36], are based on the observation that each iteration body consists of Euclidean division. It is shown that these architectures can be efficiently implemented in Field Programmable Gate Arrays (FPGA) for division by small constants or products of small constants, in comparison to other architectures, such as the Binary Tree Constant Division, and the multiplication by the reciprocal ones. The implementation of these architectures is serial, operating in a number of clock cycles that is proportional to the number of bits of the divisor. By unrolling the control loop, a combinational circuit can be obtained, but in practice, registers have to be introduced for pipelining the operation of the divider. Moreover, the table size to implement the body of the operation grows exponentially with the size of the constant, which makes linear architectures suitable only for small constant divisors.

In this paper, we propose an algorithm for hardware integer division by a constant, for which the divisor can take any constant value. Based on combinational logic, its implementation does not store intermediate results. It only uses adders and encoders. Encoders represent systems of Boolean functions. By splitting Boolean functions, the size of the tables does not grow with the size of the constant divisor, which makes the solution scalable. Thus, the proposed architectures are suitable for arbitrary integer values, both for dividends and divisors, and output both the quotient and the residue.

This paper is organised as follows. Section II introduces the

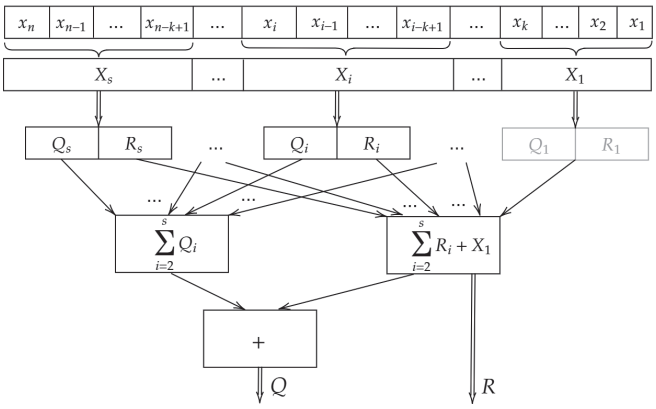

Fig. 1: Common scheme of the division.

proposed method for division, with a detailed explanation, an example, and the procedure for division including a Verilog list for its implementation. The implementation is discussed in Section III. The experimental results obtained on FPGA are discussed in Section IV, followed by conclusions and a brief outline of future work.

II. The division method

Let's consider for the division the dividend X and the constant divider d; the output are the quotient Q and the residue R, as expressed in (1), for X, d, Q, and R integers.

$$X = Q \cdot d + R \; ; \quad \frac{X}{d} = \{Q, R\} \; ; \quad \left\lfloor \frac{X}{d} \right\rfloor = Q. \qquad (1)$$

The common scheme of the division is represented in Fig. 1. Every digit of X is divided by d, which results in a quotient and a residue. The sum of all quotients, plus the quotient of the division of the sum of all residues gives the final quotient value; the remainder is the final residue value. Note that, in order to reduce hardware, X_1 (the least significant bits of the dividend $(x_k, x_{k-1}, \ldots, x_1)$) doesn't divide since it consists of the same number of bits as the divisor, thus the quotient Q_1 equals 0 or 1, and X_1 is added to other residues $R_2, R_3, \ldots, R_k$.

Let us consider, as an example, $X = 128 = 8 + 20 + 100$ and $d = 7$. The quotient of the division of 8 by 7 is 1 and the residue is 1; the quotient of the division of 20 by 7 is 2 and the residue is 6; the quotient of the division of 100 by 7 is 14 and the residue is 2. The sum of intermediate quotients is $1 + 2 + 14 = 17$ and the quotient of the division of the sum of residues, $1 + 6 + 2$, is equals to 1 and the residue is 2. Thus, the quotient of the division equals $17 + 1 = 18$ and the residue is 2.

The main steps of the proposed method for division by a constant are the following;

- the target hardware technology, ASIC or FPGA is specified;
- the dividend is split;
- the partial quotients are calculated;
- the minimization and representation of the Boolean function are performed;
- the digital circuit for implementing the divider is designed.

Let's illustrate the mentioned five-step procedure of division on an example of the division of 16-bit integers by $d = 13$, targeting an FPGA Xilinx Virtex-7.

A. Target hardware

The target hardware defines the way the Boolean minimization is performed:

- custom library for ASIC entails the implementation of either binary decision diagram minimization, two-level minimization, or other minimizations according to the custom library characteristics;
- number of look-up-tables (LUTs) and inputs of the LUTs of the targeted FPGA defines the type of multi-level Boolean minimization.

As Virtex-7 is equipped with 6-input LUTs, we consider 6-input Boolean functions. The encoders of the divider architecture are systems of minimized Boolean functions. Thus, minimization is crucial on the required number of LUTs on FPGA, and on the required number of logic elements on ASIC.

B. Splitting dividend

The n-bit dividend X is split into $s = \lceil \frac{n}{k} \rceil$ sub-vectors, where k is the number of bits of the divisor d. In binary, multiplication by 2^k is equivalent to the concatenation of k zeros as the least significant bits. For instance, multiplication of (10101) (21 in decimal) by 2^6 is $(10101\underbrace{000000}_{\text{k=6}})$ (1344 in decimal).

Generalizing this idea, any integer can be split into a set of bit sub-vectors, in this case, we split a dividend into sub-vectors with the same range as the divisor (k). For instance, 16-bit $X = (x_{16}, x_{15}, \cdots, x_1)$ equals to (1101101010111001) (in decimal 55993) can be split into four 4-bit sub-vectors $X = (X_4, X_3, X_2, X_1)$:

$$\begin{aligned}
&(\underbrace{1101}_{X_4}\underbrace{1010}_{X_3}\underbrace{1011}_{X_2}\underbrace{1001}_{X_1}) = \\
&X_1 + 2^4 \cdot X_2 + 2^8 \cdot X_3 + 2^{12} \cdot X_4 = \\
&(1001) + 2^4(1011) + 2^8(1010) + 2^{12}(1101) = \\
&(1001) + (10110000) + (101000000000) + \\
&(1101000000000000).
\end{aligned}$$

C. Partial quotients calculation

The next step of the proposed method is to define quotients $Q_1, Q_2, \ldots, Q_s$ and residues $R_1, R_2, \ldots, R_s$ from the division of every sub-vector $X_1, X_2, \ldots, X_s$ by the constant d, respectively, as defined in (2), with $i = \overline{1, s}$ and $Q_i^{max} = max\{Q_i\}$.

$$\frac{X_i}{d} = \{Q_i^{max}, R_i\} \; , \quad \left\lfloor \frac{X_i}{d} \right\rfloor = Q_i^{max} \; . \qquad (2)$$

The aim of splitting the dividend into sub-vectors is to represent the result of the division base on s truth tables, where the i-th table ($i = \overline{1, s}$) represents sub-quotients Q_i from the division of sub-dividends X_i by divider d. Thus,

the calculation of sub-quotient can be achieved as a system of j-Boolean functions with k inputs (variables), where j is bit-range of $max\{Q_i\}$, i.e.

$$Q_i = Q_i(X_i)\ ,\quad R_i = R_i(X_i). \tag{3}$$

For the previous example, $X = (x_{16}, x_{15}, \cdots, x_1)$ and $d = 13$:

- $max\{Q_1\} = 1$, when $X_1 = 2^0 \cdot x_1 + 2^1 \cdot x_2 + 2^2 \cdot x_3 + 2^3 \cdot x_4 > 12$, and the greatest residue among all possible R_1 is 12 (if $x_3 = x_4 = 1$ and $x_1 = x_2 = 0$) (this step is not included in the hardware architecture, because adding X_1 to other residues is more efficient than calculating Q_1 and R_1, as shown next);
- $max\{Q_2\} = 18$ (5-bit number), when $X_2 = 2^4 \cdot x_5 + 2^5 \cdot x_6 + 2^6 \cdot x_7 + 2^7 \cdot x_8 > 233$ (see the last line of Table II), and the greatest residue among all possible R_2 is 12 (if $x_7 = 1$ and $x_5 = x_6 = x_8 = 0$; see the fifth line of Table II, highlighted gray color);
- $max\{Q_3\} = 295$ (9-bit number), when $X_3 = 2^8 \cdot x_9 + 2^9 \cdot x_{10} + 2^{10} \cdot x_{11} + 2^{11} \cdot x_{12} > 3834$ (see the last line of Table III), and the greatest residue among all possible R_3 is 12 (if $x_{10} = x_{12} = 1$ and $x_9 = x_{11} = 0$; see the eleventh line of Table III, highlighted gray color);
- $max\{Q_4\} = 4726$ (13-bit number), when $X_4 = 2^{12} \cdot x_{13} + 2^{13} \cdot x_{14} + 2^{14} \cdot x_{15} + 2^{15} \cdot x_{16} > 61437$ (see the last line of Table IV), and the greatest residue among all possible R_4 is 12 (if $x_{15} = x_{16} = 1$ and $x_{13} = x_{14} = 0$; see the thirteenth line of Table IV, highlighted gray color).

D. Boolean Functions Minimization and Synthesis

The technique of Boolean minimization critically influences the hardware cost and the critical path of the division. Tables II, III, IV consist of 2^k lines, k columns of inputs, j columns of outputs for Q_i and k columns for R_i. The resulting systems of Boolean functions are minimized with two-level or multi-level minimization approaches, for example by applying Espresso [34] and ABC [35] tools. This research is focused on the division for FPGA with 6-input LUTs, thus performing multi-level minimization for 6-input LUTs in ABC: *fpga -K 6*.

For the example provided in this paper, a 16-bit dividend X and 4-bit divisor d, the number of bits of the sub-dividend and sub-quotients is presented in Table I. As far as the result of the division of Q_1 by the k-bit divisor is 1-bit number ($Q_1 = 1$ if $X_1 - d \geq 0$, else $Q_1 = 0$) and k-bit residue, thus the generation of the truth table for Q_1 and R_1 is omitted (as already mentioned in the previous subsection).

E. Divider architecture

The proposed divider has two sorts of elements: *subcoder* (SC) and adders (S). The subcoders transform sub-dividend into sub-quotients and residues. The adders calculate the sum of sub-quotients. The scheme for division that supports the proposal is the one already presented in Fig. 1.

TABLE I: Sub-dividends and sub-quotients.

X		
X_1	$x_4\ x_3\ x_2\ x_1$	(4 bits)
X_2	$x_8\ x_7\ x_6\ x_5$	(4 bits)
X_3	$x_{12}\ x_{11}\ x_{10}\ x_9$	(4 bits)
X_4	$x_{16}\ x_{15}\ x_{14}\ x_{13}$	(4 bits)
Q		
Q_2	Q_1^2	$q_4^2, q_3^2, q_2^2, q_1^2$ (4 bits)
	Q_2^2	q_5^2 (1 bit)
Q_3	Q_1^3	$q_4^3, q_3^3, q_2^3, q_1^3$ (4 bits)
	Q_2^3	$q_8^3, q_7^3, q_6^3, q_5^3$ (4 bits)
	Q_3^3	q_9^3 (1 bit)
Q_4	Q_1^4	$q_4^4, q_3^4, q_2^4, q_1^4$ (4 bits)
	Q_2^4	$q_8^4, q_7^4, q_6^4, q_5^4$ (4 bits)
	Q_3^4	$q_{12}^4, q_{11}^4, q_{10}^4, q_9^4$ (4 bits)
	Q_3^4	q_{13}^4 (1 bits)

TABLE II: Truth table for Q_2 and R_2.

X_2	Q_2	R_2
$x_8 x_7 x_6 x_5$	$Q_2^2\ Q_1^2$	
0 0 0 0	0 0000	0000
0 0 0 1	0 0001	0011
0 0 1 0	0 0010	0110
0 0 1 1	0 0011	1001
0 1 0 0	0 0100	1100
0 1 0 1	0 0110	0010
0 1 1 0	0 0111	0101
0 1 1 1	0 1000	1000
1 0 0 0	0 1001	1011
1 0 0 1	0 1011	0001
1 0 1 0	0 1100	0100
1 0 1 1	0 1101	0111
1 1 0 0	0 1110	1010
1 1 0 1	1 0000	0000
1 1 1 0	1 0001	0011
1 1 1 1	1 0010	0110

Let's continue the explanation with the referred example, and highlight in the truth tables the binary subvectors with the highest value in the following way:

- by yellow the sub-quotients $max\{Q_1^2\} = max\{Q_1^3\} = max\{Q_1^4\} = (1110)$;
- by green sub-quotients $max\{Q_2^2\} = (1)$ and $max\{Q_2^3\} = max\{Q_2^4\} = (1110)$;
- by orange sub-quotients $max\{Q_3^3\} = (1)$ and $Q_3^4(1110)$;
- by blue sub-quotients $max\{Q_4^4\} = (1)$;

The proposed architecture for division is depicted in Fig. 2, where purple and red elements calculate quotient and residue at the same time; only red elements are required to calculate the residue. The procedure for division (X) by a constant (d) is presented in Algorithm 1. It is easy understandable as a generalization of the division method described for the example of the dividend X with 16 bits and the divisor the constant value $d = 13$. The first sub-coder

TABLE III: Truth table for Q_3 and R_3.

X_3	Q_3	R_3
$x_{12}x_{11}x_{10}x_9$	Q_3^3 Q_2^3 Q_1^3	
0 0 0 0	0 0000 0000	0000
0 0 0 1	0 0001 0011	1001
0 0 1 0	0 0010 0111	0101
0 0 1 1	0 0011 1011	0001
0 1 0 0	0 0100 1110	1010
0 1 0 1	0 0110 0010	0110
0 1 1 0	0 0111 0110	0010
0 1 1 1	0 1000 1001	1011
1 0 0 0	0 1001 1101	0111
1 0 0 1	0 1011 0001	0011
1 0 1 0	0 1100 0100	1100
1 0 1 1	0 1101 1000	1000
1 1 0 0	0 1110 1100	0100
1 1 0 1	1 0000 0000	0000
1 1 1 0	1 0001 0011	1001
1 1 1 1	1 0010 0111	0101

TABLE IV: Truth table for Q_4 and R_4.

X_4	Q_4	R_4
$x_{16}x_{15}x_{14}x_{13}$	Q_4^4 Q_3^4 Q_2^4 Q_1^4	
0 0 0 0	0 0000 0000 0000	0000
0 0 0 1	0 0001 0011 1011	0001
0 0 1 0	0 0010 0111 0110	0010
0 0 1 1	0 0011 1011 0001	0011
0 1 0 0	0 0100 1110 1100	0100
0 1 0 1	0 0110 0010 0111	0101
0 1 1 0	0 0111 0110 0010	0110
0 1 1 1	0 1000 1001 1101	0111
1 0 0 0	0 1001 1101 1000	1000
1 0 0 1	0 1011 0001 0011	1001
1 0 1 0	0 1100 0100 1110	1010
1 0 1 1	0 1101 1000 1001	1011
1 1 0 0	0 1110 1100 0100	1100
1 1 0 1	1 0000 0000 0000	0000
1 1 1 0	1 0001 0011 1011	0001
1 1 1 1	1 0010 0111 0110	0010

SC_1 calculates Q_2 and R_2, according to Table II; the second sub-coder SC_2, calculates Q_3 and R_3, according to Table III; the third sub-coder SC_3 calculates Q_3 and R_3, according to Table IV.

The first adder (S_1) calculates $X_1 + R_2 + R_3 + R_4 = S_1$. As we mentioned, $max\{X_1\} = (1111)$ and $max\{R_2\} = max\{R_3\} = max\{R_4\} = (1100)$, i.e. $S_1 \leq 15 + 12 + 12 + 12 = 51$ (6-bit number). Note, that:

- the outputs of S_1 are the inputs of the sub-coder SC_4. This sub-coder implements the truth table with 6 input columns $s_6^1, s_5^1, s_4^1, s_3^1, s_2^1, s_1^1$, 52 lines and 6 output columns, as represented on Table V;
- the four right most columns of Table V represent not only the residue from the division of S_1 by 13, but also the general residue $R = (r_4, r_3, r_2, r_1)$ from the division of X by 13 – R;
- the two most significant output bits of SC_4 $QR = (qr_2, qr_1)$ represent the quotient from the division of S_1 by 13.

The inputs of the adder S_2 include 4-bit Q_1^2, Q_1^3, Q_1^4 and

TABLE V: Truth table for Q_4 and R.

S_1	QR	R
$s_6^1 s_5^1 s_4^1 s_3^1 s_2^1 s_1^1$	$qr_2 qr_1$	
0 0 0 0 0 0	0 0	0000
0 0 0 0 0 1	0 0	0001
0 0 0 0 1 0	0 0	0010
...	...	...
0 1 1 0 1 0	1 0	0000
...	...	...
1 0 0 1 1 0	1 0	1100
...	...	...
1 1 0 0 1 0	1 1	1011
1 1 0 0 1 1	1 1	1100

the output S_2 is the sum. According to the truth tables $max\{Q_1^2\} = max\{Q_1^3\} = max\{Q_1^4\} = 14$ (see yellow sub-quotients). Thus $max\{S_2\} = 14 + 14 + 14 = 42$ (6-bit number $S_2 = (s_6^2, s_5^2, s_4^2, s_3^2, s_2^2, s_1^2)$.

The inputs of the adder S_3 include 1-bit Q_2^2, 4-bit Q_2^3, Q_2^4 and 2-bit (s_6^2, s_5^2) carried from $S2$: $(s_6^2, s_5^2) + Q_2^2 + Q_2^3 + Q_2^4 = S_2$. According to the truth tables $max\{Q_2^2\} = 1$ and $max\{Q_2^3\} = max\{Q_2^4\} = 14$ (see green sub-quotients). Since $max\{S_2\} = 42$, thus $max\{(s_6^2, s_5^2)\} = (10)$, and $max\{S_3\} = (s_6^2, s_5^2) + 1 + 14 + 14 = 31$ (5-bit number $S_3 = (s_5^3, s_4^3, s_3^3, s_2^3, s_1^3)$.

The inputs of the adder S_4 include 1-bit Q_3^3, 4-bit Q_3^4 and 1-bit s_5^3 carry out from $S3$: $(s_5^3) + Q_3^3 + Q_3^4 = S_4$. According to the truth tables $max\{Q_3^3\} = 1$ and $max\{Q_3^4\} = 14$ (see orange sub-quotients). Since s_5^3 is 1-bit number, thus $max\{(s_5^3)\} = (1)$, but the case s_5^3 and Q_3^3 equals 1 can not be achieved. Thus, $max\{S_4\} = 15$ (4-bit number $S_4 = (s_4^4, s_3^4, s_2^4, s_1^4)$.

The adder S_5 calculates the final value of the quotient, by adding the intermediate quotient QT and the 2-bit output RQ of the sub-coder SC_4. This sub-coder calculates the value of "accumulated" sub-residues. The 13-bit output of the fifth adder consists of the addition of the four least significant bits of S_2, S_3, S_4 and Q_4^4.

Generally, the scheme of the n-bit divider by k-bit constant includes $s = \lceil \frac{n}{k} \rceil$ subcoders (SCs) and $s + 1$ chain of adders (Ss). The key role in the efficiency of the divider is played by the interconnection between elements and the way the minimization is performed. This article explores the design on FPGA. The main cell of an FPGA is the LUT. Current FPGAs provide 5- and 6-input LUTs. LUTs allow truth tables of Boolean functions to be implemented. It means that a scheme of the divider by 5-bit and 6-bit divisor can be designed without Boolean functions minimization, i.e. SCs can be described by full disjunctive normal forms. The way of splitting Boolean functions for 7-bit and more is significantly more important than the minimization for the design on FPGA. Designing for ASICs is strongly related to the custom library of elements. In this case, the crucial role for the efficiency of a scheme belongs to the minimization or representation of Boolean functions.

The second key point of the design of an efficient architecture is the interconnection between subcoders and adders, and

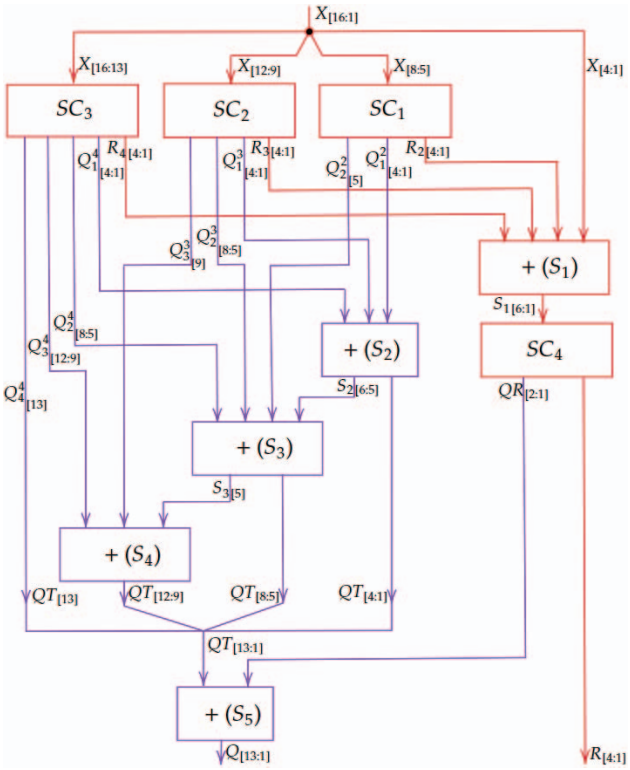

Fig. 2: Logic scheme of 16-bit divider by 13.

between adders. As was mentioned, there are in general s SCs and a chain of $s+1$ adders. The Xilinx tools automatically insert a 13-bit adder for S_5 in Fig. 2, nevertheless, QT is 13-bit and QR is 2-bit numbers. As a result, most probably Xilinx tools automatically design the scheme of S_5 with half-adders and 12 full adders. This redundancy can be fixed by reducing the chain of adders or interconnections at the output of SC_4 with S_3 or S_4. This upgrade reduces the scheme of the whole divisor and reduces area costs. This type of optimization of the divisor architecture and micro-architecture of adders and subcoders is appropriate, as referred for this particular example. The optimization is specific, according to the bit range of the dividend and the value of the divisor.

In this paper, we do not optimize the architecture and the micro-architecture but just declare a new common approach to division. The Boolean minimization and uniqueness of the data-path optimization make harder the generalization of the proposed approach for design automation.

III. Implementation

The implementation involves also the place and routing process. It divides the whole logic circuit into sub-blocks that can fit into the FPGA logic blocks, such as combinational logic blocks (CLB, which consists of LUTs), input-output blocks (IOs), block-RAMs, block multipliers, DSP, etc. The implementation places the circuit into logic blocks according to the constraints and connects the logic blocks. The performance is

Algorithm 1 Algorithm for division of X by d.

Input:
$X = (x_n, x_{n-1}, ..., x_1)$ - dividend
d - divider
Q - quotient of the division of X by d
R - residue from the division of X by d

Calculations:

1.
$BR(X) = n$ - BR returns bit-range of X
$BR(d) = k$
$p = \left\lceil \frac{n}{k} \right\rceil$
$SP(X, k, p) = (X_p, X_{p-1}, \ldots, X_1)$ - SP splits X into p k-bit sub-vectors
2.
$SC(X_2) = \frac{X_2}{d} = \{Q^2, R_2\}$ - SC returns the quotient Q^2 and the residue R_2 of the division of X_2 by d
$SC(X_3) = \{Q^3, R_3\}$
$\ldots$
$SC(X_i) = \{Q^i, R_i\}$
$\ldots$
$SC(X_p) = \{Q^p, R_p\}$
3.
$SP(Q^2, k, p_1) = (Q^2_{p_1}, Q^2_{p_1-1}, \cdots, Q^2_1)$, where SP splits sub-quotient Q^2 into p_1 k-bit vectors
$SP(Q^3, k, p_2) = (Q^3_{p_2}, Q^3_{p_2-1}, \cdots, Q^3_1)$
$\ldots$
$SP(Q^i, k, p_{i-1}) = (Q^i_{p_{i-1}}, Q^i_{p_{i-1}-1}, \cdots, Q^i_1)$
$\ldots$
$SP(Q^p, k, p_{p-1}) = (Q^p_{p_{p-1}}, Q^p_{p_{p-1}-1}, \cdots, Q^p_1)$
4.
$S_1 = X_1 + R_2 + \cdots + R_p$
$SC(S_1) = \{QR, R\}$ - R is the residue of the division
5.
$S_2 = QR + \sum_{j=2}^{p} Q_1^j$
$S_3 = S_2[BR(S_2) : k+1] + \sum_{j=2}^{p} Q_2^j$, where $S_2[BR(S_2 : k+1)]$ is the most significant $BR(S_2) - k$ bits of S_2
$\ldots$
$S_i = S_{i-1}[BR(S_{i-1}) : k+1] + \sum_{j=2}^{p} Q_{i-1}^j$,
if $j < i-1$ then $Q_{i-1}^j = 0$
$\ldots$
$S_p = S_{p-1}[BR(S_{p-1}) : k+1] + \sum_{j=2}^{p} Q_{p-1}^j$
$S_{p+1} = S_p[BR(S_p) : k+1] + Q_p^p$
Output
$Q = (S_{p+1} \& S_p[k:1] \& \cdots \& S_2[k:1])$, where
$\&$ means concatenation
R has been calculated on the step **4.**

defined by the total delay time which can be calculated as (4).

$$\begin{aligned} &total\ delay\ = \\ &data\ path\ (logic\ delay) + \ routing\ path\ delay + \\ &\quad collateral\ path\ (skew,\ uncertainty,\ etc.). \end{aligned} \tag{4}$$

The implementation maps the architecture of the divider into LUTs. It means that logic elements of the proposed architecture, adders and subcoders, are automatically represented in systems of a Boolean function. For instance, full and half-adder will be placed into the same hardware resources on the FPGA, despite the scheme of the half-adder consists of two $2-$input logic elements while the full adder includes five $2-$input logic elements. Thus, it is not relevant which type of adder architecture is adopted, such as ripple-carry or carry-lookahead, as an optimal form of representation of Boolean functions. The attention is focused on features of FPGA, such as the number of inputs of the LUT, tipycally 4, 5, and 6, and interconnection between groups of elements, rather than the micro-architecture of basic elements. The implementation of the proposed approach is scalable for an arbitrary number of LUTs' inputs, number of inputs of subcoders, for values of dividends and divisors. In this paper, we consider 6-input LUTs and 6-input subcoders.

Let us suppose that the target FPGA is equipped with $3-$input LUTs. In this system of Boolean functions for Q_2 and R_2 (see Table II), which describes SC_1 and depends on four variables x_5, x_6, x_7, x_8, might be represented as the composition of functions (ABC tool [35] has been applied for splitting the Table II) dependent on three variables (5).

$$\begin{gathered} q_5^2 = x_8 \cdot x_7 \cdot \overline{u_1}; \qquad q_4^2 = (x_7 \cdot \overline{u_8}) \vee (x_8 \cdot \overline{x_7}); \\ q_3^2 = (\overline{u_2} \cdot u_3) \vee \overline{u_4}; \qquad q_2^2 = (x_7 \cdot \overline{u_5}) \vee (\overline{x_7} \cdot u_6); \\ q_1^2 = u_{11} \cdot u_{12} \vee \overline{u_{13}}; \end{gathered} \tag{5}$$

$$\begin{gathered} r_4^2 = \overline{u_8} \vee u_9; \qquad r_3^2 = x_6 \cdot \overline{u_{10}} \vee u_9; \\ r_2^2 = (\overline{x_7} \cdot \overline{u_6}) \vee (x_7 \cdot \overline{u_7}); \qquad r_1^2 = q_1^2, \end{gathered}$$

where

$$\begin{gathered} u_1 = \overline{x_5} \cdot \overline{x_6}; \qquad u_2 = (x_6 \cdot x_8) \vee (\overline{x_6} \cdot \overline{x_8}); \\ u_3 = \overline{x_5} \cdot x_7; \qquad u_4 = (x_6 \vee \overline{x_7} \vee x_8) \cdot (\overline{x_6} \vee x_7 \vee \overline{x_8}); \\ u_5 = x_8 \cdot ((\overline{x_5} \cdot x_6) \vee (x_5 \cdot \overline{x_6})) \vee \overline{x_8} \cdot ((\overline{x_5} \cdot \overline{x_6}) \vee (x_5 \cdot x_6)); \\ u_6 = (\overline{x_6} \vee x_8) \cdot (\overline{x_5} \vee x_6 \vee \overline{x_8}); \\ u_7 = (x_5 \cdot \overline{x_6} \cdot x_8) \vee (\overline{x_8} \cdot (\overline{x_5} \vee x_6)); \\ u_8 = (x_5 \vee x_6 \vee \overline{x_8}) \cdot (\overline{x_5} \vee \overline{x_6} \vee x_8); \\ u_9 = x_7 \cdot \overline{x_8} \cdot u_{11}; \qquad u_{10} = (x_5 \cdot \overline{u_8}) \vee (\overline{x_5} \cdot x_7 \cdot x_8); \\ u_{11} = \overline{x_5} \cdot \overline{x_6}; \qquad u_{12} = \overline{x_7} \cdot x_8; \\ u_{13} = (\overline{x_5} \vee x_7) \cdot (x_5 \vee \overline{x_6} \vee \overline{x_6}). \end{gathered}$$

Thus, the implementation maps the scheme of SC_1, for calculating Q_2 and R_2, as shown in Fig. 3.

IV. Experimental results

This section provides experimental results to evaluate the proposed approach on FPGAs, equipped 6-input LUTs, and compares these results with the state-of-the-art approaches [1].

[1]Verilog-files prepared for these experiments can be found in [38].

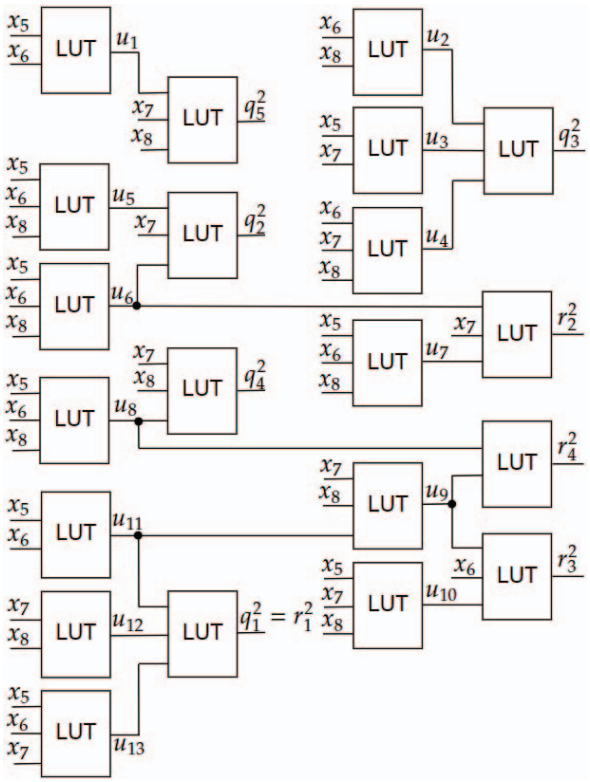

Fig. 3: LUTs realization of Q_2 and R_2.

-max_bram*	0
-max_uram*	0
-max_dsp*	0
-max_bram_cascade_height*	0
-max_uram_cascade_height*	0

Fig. 4: Synthesis settings.

Performance, the critical path as total delay of the implementation, and the circuit area (LUTs) are presented, for the Xilinx Vivado version 2019.1.

A. Evaluating dividers: proposed vs generated with Vivado

This subsection compares the proposed structures to the ones automatically generated with Vivado for Virtex-7 (xc7v585tffg1157-3). The modeling was performed for three prime dividers $d = 47, 113, 241$ (6-, 7- and 8-bit numbers, respectively) and for n-bit ranges of X, where $n = 24$, 36, 48, and 60 bits. Performance is measured as the critical path from pin-to-pin (from inputs to outputs) on FPGA. The prime numbers for the divisor are typically used, for example in residue number systems.

An FPGA is typically equipped with embedded elements such block-RAMs, DSPs, multipliers, and others. To compare Xilinx division with the proposed approach, we exclude the usage of embedded elements and utilize only CLBs (LUTs) and IOs (see Fig. 4) in this section.

The proposed approach of division (*proposed* in Table VI) has been designed using Verilog and compared in this section

TABLE VI: Quotient and residue calculation on Virtex-7.

		Proposed		Vivado	
d	n	Delay ns	Area #LUTs	Delay ns	Area #LUTs
47	24	9.4	126	12.7	407
	36	15.3	258	20.2	780
	48	21.3	395	17.9	1145
	60	27.7	483	19.5	1642
113	24	10.0	221	12.8	434
	36	14.0	507	14.3	830
	48	20.0	496	18.3	1384
	60	27.3	694	17.6	2401
241	24	11.1	248	13.8	549
	36	12.8	471	15.4	722
	48	16.6	729	18.7	1756
	60	23.6	763	19.8	1941

with Xilinx functions division and modulus at the same time (*Vivado* in Table VI):

$$assign\ Q\ =\ X\ /\ d;\quad assign\ R\ =\ X\ \%\ d.$$

The proposed division approach shows a reduction on the number of LUTs compared with embedded algorithms in Xilinx for:

- $d = 47$, in more than 3x for the considered values of n;
- $d = 113$, between 1.6x for $n = 36$, and 3.5x for $n = 60$;
- $d = 241$, between 1.6x for $n = 36$, and 2.5x times for $n = 60$.

The proposed approach speedup the division:

- up to 20% for 24-bit dividend and all considered divisors;
- 25% for $d = 47$ and on 17% for 241 for 36-bit dividend;
- 11% for $d = 241$ add 48-bit dividend.

We conducted experiments for up to 100-bit for X and up to 12-bit divisors. Experimental results show up to three times reduction in area costs compared with the Xilinx approach, and equivalent performance for up to 48-bit dividends.

B. Evaluating dividers: proposed vs state-of-the-art

This subsection compares the proposed dividers' architectures with state-of-the-art hardware architectures for divisors by constants [37], both implemented on the Kintex-7. The modeling was performed for four dividers $d = 3, 5, 11, 23$ (2-, 3-, 4- and 5-bit numbers, respectively), and three bit-ranges for X, $n = 16, 32$, and 64 bits. In order to compare the area and the delay, the results provided in [37], for the architecture based on multiplication by the reciprocal (Recip), the Binary Tree Constant Divider (BTCD), and the Linear Architecture (LinArch), are herein directly presented in Tables VII and VIII.

Table VII provides the results of the architectures to calculate quotients and residues for $d = 3, 5, 11, 23$, where *BR* means the number of Block-RAMs. Table VIII provides the results of the architectures to calculate only residues for $d = 3, 5$ and the LinArch and the BTCD architectures, and for $d = 3, 23$ and the Recip architecture.

TABLE VII: Quotient and residue calculation on Kintex-7, comparison with the state-of-the-art [37].

		Proposed		Vivado		LinArch [37]		BTCD [37]		Recip [37]	
d	n	Delay ns	Area #LUTs	Delay ns	Area #LUTs	Delay ns	Area #LUTs	Delay ns	Area #LUTs	Delay ns	Area #LUTs
3	16	4.1	40	7.3	180	3.6	17	3.7	37	4.5	52
	32	11.4	98	11.8	612	6.0	32	4.8	95	6.1	139
	64	27.5	379	17.5	2004	13.5	63	6.2	225	8.5	346
5	16	4.0	52	4.0	31	4.4	21	3.8	44	4.9	54
	32	10.6	123	10.4	719	9.3	45	4.7	109	7.0	140
	64	27.3	386	15.9	2311	20.1	93	6.7	270	8.6	346
11	16	3.9	53	6.1	40	8.0	39	3.8	79	4.8	104
	32	10.7	159	9.8	566	17.9	87	6.1	212	6.6	219
	64	26.9	436	15.1	2052	39.0	183	8.8	526	8.4	503
23	16	3.9	52	7.1	129	7.4	69	5.6	197	5.1	83
	32	10.4	187	10.2	445	18.5	165	6.8	1 BR +436	7.3	169
	64	26.1	493	14.4	1542	36.6	357	6.5	2.5 BR +959	9.0	401

TABLE VIII: Only residue calculation on Kintex-7, comparison with the state-of-the-art [37].

		Proposed		Vivado		LinArch [37]		BTCD [37]		Recip [37]	
d	n	ns	LUTs	ns	LUTs	ns	LUTs	ns	LUTs	ns	LUTs
3	16	3.6	9	5.4	57	3.5	5	3.7	5	4.2	51
	32	3.9	21	7.5	191	6.5	11	3.8	12	6.0	138
	64	4.3	47	10.2	637	14.2	21	4.8	25	7.9	345
5	16	3.6	21	3.7	19	4.4	14	3.6	14	–	–
	32	4.0	33	7.7	259	9.3	45	3.7	29	–	–
	64	4.8	66	9.4	779	20.1	93	4.5	62	–	–
23	16	3.7	23	5.5	49	–	–	–	–	3.8	71
	32	5.1	74	7.3	159	–	–	–	–	5.8	158
	64	5.9	151	8.9	536	–	–	–	–	7.9	390

According to the experimental results, the proposed approach provides the best trade-off for the five considered approaches with significant advantages:

- performance and area cost of the proposed architecture for $d = 23$ and $n = 16$ and quotient-residue calculation is superior to all the others;
- performance and area costs of the proposed architecture are always better for $d = 23$, and any value of n for residue-only calculation, which means that the proposed architecture is scalable, for larger values of d;
- in comparison with the LinArch, the delay associated with the proposed architecture is significantly less when the value of the constant d increases;
- as expected, for larger values of the constant ($d = 23$), the BTCD architecture exhibits less delay but at the cost of a significantly larger circuit area.

V. Conclusions and further work

This paper proposed an algorithm for hardware integer division by a constant. The most distinguishing feature is that the algorithm is scalable for an arbitrary value of dividend (X) and an arbitrary value of divisor (d). The limitation of the divisor depends on the ability to minimize systems of Boolean functions. The proposed approach is based only on combination logic, adders, and encoders, which represent systems of Boolean functions. The Boolean minimization approach significantly influences the hardware cost and the critical path of the scheme. Experimental results show the superiority of the proposal, in comparison to the Xilinx Vivado hardware divisor automatically generated by the Xilinx Vivado and the state-of-the-art divisors by constants. It is shown that the proposed approach is scalable, is more efficient than the others for larger values of the constant divisor, and provides a good trade-off between performance and cost. This paper does not consider the optimization of the adders tree in the architecture of the divisor. This is a topic for future research in the direction this paper opened for hardware division by constants.

Acknowledgment

The authors would like to express sincere thanks to anonymous reviewers, which allow them to significantly improve the earlier version of this paper. This work was partially supported by FCT (Fundação para a Ciência e a Tecnologia, Portugal), through the UIDB/50021/2020 project, and EuroHPC Joint Undertaking through grant agreement No 956213 (SparCity).

References

[1] N. Takagi, S. Kadowaki, K. Takagi, "A hardware algorithm for integer division", 17th IEEE Symposium on Computer Arithmetic (ARITH'05), 2005, Cape Cod, MA, USA, 27-29 June, 2005.

[2] U.S. Patankar, M.E. Flores, A. Koel, "Novel data dependent divider circuit block implementation for complex division and area critical applications", Scientific Reports, Springer Nature, 2023.

[3] U.S. Patankar, A. Koel, "Review of basic classes of dividers based on division algorithm", IEEE Open Access, Vol. 9, 2021.

[4] A. Yasin, T. Su, S. Pillement, M. Ciesielski, "Functional verification of hardware dividers using algebraic model", 2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC), Cuzco, Peru, 06-09 Oct. 2019.

[5] E. M. Clarke, S. M. German, X. Zhao, "Verifying the SRT division algorithm using theorem proving techniques", in Computer Aided Verification (Lecture Notes in Computer Science), vol. 1102, R. Alur and T. A. Henzinger, Eds. Berlin, Germany: Springer, 1996, pp. 111-122,

[6] D. Piso, J.A. Pineiro, J.D. Bruguera, "Analysis of the impact of different methods for division/square root computation in the performance of a superscalar microprocessor", Proc. Euromicro Symposium on Digital System Design. Architectures, Methods and Tools, 2002, pp. 218–225.

[7] P. Sinha, "Smart Sensors Use DSCs for Embedded Signal Processing", Microchip Technology Inc., 2021.

[8] I. Koren, "Computer Arithmetic Algorithms", 2nd ed., A K Peters, Natick, Massachusets, 2002.

[9] K. Tatas, D. J. Soudris, D. Siomos, M. Dasygenis, A. Thanailakis, "A novel division algorithm for parallel and sequential processing", Proc. 9th Int. Conf. Electron., Circuits, Syst., Dubrovnik, Croatia, Sep. 2002, pp-553-556.

[10] D. G. Bailey, "Space eficient division on FPGAs", Proc. Electron. New Zealand Conf., 2006, pp. 206-211.

[11] S. F. Obermann, M. J. Flynn, "Division algorithms and implementations", IEEE Trans. Comput., vol. 46, no. 8, pp. 833-854, Aug. 1997.

[12] S. Dixit, M. Nadeem, "FPGA accomplishment of a 16-bit divider", Imperial J. Interdiscipl. Res., vol. 3, no. 2, pp. 140-143, 2017.

[13] M. F. Kasim, T. Adiono, M. F. Zakiy, M. Fahreza, "FPGA implementation of Fixed-point divider using pre-computed values", in Proc. Technol., vol. 11, Jun. 2013, pp. 206-211.

[14] R. S. Hongal, D. J. Anita, "Comparative study of different division algorithms for fixed and floating point arithmetic unit for embedded applications", Int. J. Comput. Sci. Eng., vol. 4, no. 9, pp. 48-54, 2016.

[15] N. Boullis, A. Tisserand, "On digit-recurrence division algorithms for self-timed circuits", INRIA-Institut Nat. De Recherche En Informatique Et En Automatique, France, Tech. Rep. RR-4221, Jul. 2001.

[16] J. Detrey, F.A. de Dinechin, "A Tool for unbiased comparison between logarithmic and floating-point arithmetic", VLSI Signal Process. Syst. Signal Image Video Technol. 49, 161–175 (2007).

[17] M.F. Kasim, T. Adiono, M. Fahreza, M.F. Zakiy, "FPGA implementation of fixed-point divider using pre-computed values", Procedia Technol. 11, 2013, 206–211.

[18] S.F. Oberman, M.J. Flynn, "An analysis of division algorithms and implementations", Technical Report CSL-TR-95-675 (StanfordUniversity, 1995).

[19] S. Kaur, M. Singh, and R. Agarwal, "VHDL implementation of non-restoring division algorithm using high-speed adder/subtractor", Int. J. Adv. Res. Electr., Electron. Instrum. Eng., vol. 2, no. 7, pp. 3317-3324, Jul. 2013.

[20] E. Matthews, A. Lu, Z. Fang, L. Shannon, "Rethinking integer divider design for FPGA-based soft-processors", Proc. IEEE 27th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM), Apr. 2019, pp. 289-297.

[21] B. Mehta, J. Talukdar, S. Gajjar, "High speed SRT divider for intelligent embedded system", 2017 International Conference on Soft Computing and Its Engineering Applications (icSoftComp), 2017, pp. 1–5.

[22] J. Kumari, M. Y. Yasin, "Design and Soft Implementation of N-bit SRT Divider on FPGA through VHDL", Int. J. Innov. Eng., Sci. Manage., vol. 3, no. 4, pp. 13-19, Apr. 2015.

[23] D.A. Patterson, J.L. Hennessy, "Computer organization & design", 4th ed., Morgan Kaufmann, San Francisco, California, 2009.

[24] J.L. Hennessy, D.A. Patterson, "Computer architecture. A quantitative approach", 5th ed., Morgan Kaufmann, San Francisco, California, 2012.

[25] M. Lu, "Arithmetic and logic in computer systems", Wiley-Interscience, Hoboken, New Jersey, 2004.

[26] S.L. Harris, D.M. Harris, "Digital design and computer architecture", Elselvier, 2016.

[27] R. Trummer, P. Zinterhof, R. Trobec, "A high-performance data-dependent hardware divider", Systems and Simulation, Parallel Numerics. Ljubljana, Slovenia: Salzburg Univ.; Ljubljana Josef Stefan Institute, 2005, ch. 7, pp. 193-206.

[28] T Drane, W.-C. Cheung, G. Constantinides, "Correctly rounded constant integer division via multiply-add", 2012 IEEE International Symposium on Circuits and Systems (ISCAS), Seoul, South Korea, 20-23 May, 2012, pp. 1243-1246.

[29] A.R. Omondi, "Cryptography arithmetic. Algorithms and hardware architectures", Springer Nature Switzerland, 2020.

[30] W. Stallings, "Cryptography and network security: principles and practice. 7th Edition", Pearson, 2017.

[31] T. Granlund, P. Montgomery, "Division by invariant integers using multiplication", PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, Aug. 1994, pp. 61–72.

[32] J.-M. Muller, A. Tisserand, B. de Dinechin, C. Monat, "Division by constant for the ST100 DSP microprocessor", 17th IEEE Symposium on Computer Arithmetic (ARITH'05), Jun. 2005, pp. 124–130.

[33] P.V.A. Mohan, "Residue number system. Theory and applications", Springer International Publishing, 2016, 351 p.

[34] https://embedded.eecs.berkeley.edu/pubs/downloads/espresso

[35] https://people.eecs.berkeley.edu/~alanmi/abc/

[36] F. de Dinechin, L.-S. Didier, "Table-based division by small integer constants", Int. Symp. Applied Reconfigurable Computing (ARC), Lecture Notes in Computer Science, vol. 7199, pp. 53-63, Hong Kong, China, Mar. 2012.

[37] H. F. Ugurdag, F. de Dinechin, Y. S. Gener, S. Gören, L.-S. Didier, "Hardware Division by Small Integer Constants", IEEE Transactions on Computers, Vol. 66, No. 12, Dec. 2017.

[38] https://github.com/ZeboZebo702/ARITH_2023_constant_division