LUT-Oriented Boolean Decomposition for Efficient Arithmetic on FPGAs and ASICs

Danila Gorodecky, Leonel Sousa
INESC-ID, Instituto Superior Tecnico, Universidade de Lisboa
Lisbon, Portugal
danila.gorodcky@gmail.com, leonel.sousa@tecnico.ulisboa.pt

***Abstract*—This paper proposes a systematic methodology for designing efficient arithmetic units on FPGAs and ASICs through Boolean function optimization tailored for LUT architectures. These units target operations that include multiplication and division by constant, modular multiplication and reduction by using decomposition techniques that rely exclusively on combinational logic. A comprehensive experimental evaluation across Xilinx Vivado FPGA and Cadence and Synopsys ASIC toolchains demonstrates substantial improvements. FPGA results show up to a 20× reduction in LUT usage for modular reduction and an 11× reduction for special-form constant multiplication, with critical path delays reduced by up to 1.8x across operations. ASIC synthesis using Synopsys achieves up to a 100× reduction in cell count for multiplication and timing improvements of up to 30x for modular reduction. Cadence synthesis reveals platform-dependent trade-offs, yielding timing improvements of up to 5× for most operations, while area efficiency varies by operation type: modular reduction benefits from up to a 20x reduction in cell count, whereas general constant multiplication may incur up to a 2.7× increase in cell usage. Comparisons with FloPoCo indicate competitive performance, with the proposed method consistently outperforming prior work in either resource utilization or timing, and often in both metrics. These results demonstrate that the proposed methodology provides a general and effective approach for optimizing combinational arithmetic units across both FPGA and ASIC platforms.**

***Index Terms*—FPGA, ASIC, computer arithmetic, Boolean functions, logic synthesis, modular arithmetic.**

I. Introduction and State of the Art

Efficient implementation of arithmetic operations on FPGAs and ASICs requires a deep understanding of the characteristics of the target architecture and technology. In FPGAs, Look-Up Tables (LUTs) serve as the fundamental building blocks for combinational logic, and effective optimization depends on exploiting LUT structures and dual-output capabilities. For ASICs, standard cell libraries provide the implementation substrate, where the Boolean function decomposition and the technology mapping determine final circuit characteristics. Both platforms benefit from systematic approaches to arithmetic design. However, the required platform-specific optimizations differ substantially.

No universal optimal approach exists for efficiently mapping arbitrary arithmetic operations across these platforms due to the inherent diversity of operation characteristics and technologies. This paper addresses four arithmetic operations that share optimization for Electronic Design Automation (EDA) tools challenges yet require distinct implementation strategies.

Multiplication by a constant is extensively used in numerical algorithms [1], [2], signal/image/video processing, cryptography, neural networks, and residue number systems (RNS) [3]–[5]. The efficiency of this operation directly impacts digital filter performance, FFT implementations, and matrix operations [5].

Modular multiplication plays a pivotal role in digital signal processing [6], cryptography [5], and RNS [3], [5], where operations on small moduli facilitate parallel processing and improved throughput.

Modular reduction is a cornerstone operation underlying numerous computational systems. Beyond cryptography, it is fundamental to RNS implementations [3], [4], serving as the core conversion operation. While efficient techniques exist for special moduli (e.g., $2^n \pm \psi$), arbitrary moduli require more complex approaches. Until recently, modular reduction was not efficiently synthesized by EDA tools, remaining a major limitation for widespread RNS adoption.

Division by a constant occurs frequently across diverse applications [7], including counters [8], cryptography [4], [9], networks [10], general processing [11], [12], and accessing interleaved memory banks with non-power-of-two configurations. Hardware implementation poses unique challenges due to design complexity and verification requirements [13], [14].

Contemporary design tools enable system description through both traditional HDL languages and High-Level Synthesis approaches. Algorithmic descriptions often exhibit redundancy and suboptimal performance due to generalized translation of mathematical algorithms into Register Transfer Level (RTL) representations. FloPoCo (Floating-Point Cores) [15], [16] is a state-of-the-art arithmetic core generator for designing arithmetic units [17] based on high-level representations. Alternative approaches focus on hardware design using Boolean minimization tools like ABC [18], which optimizes combinational logic while considering target architecture. However, these tools do not account for both the main characteristics of arithmetic functions and the subtle architectural features, such as Xilinx FPGA LUTs operating in multiple modes.

The herein proposed approach for designing efficient arithmetic units balances universality with emphasis on the technology mapping stage, representing arithmetic operations as optimized Boolean functions specifically tailored to LUT-based architectures. This provides a systematic methodology

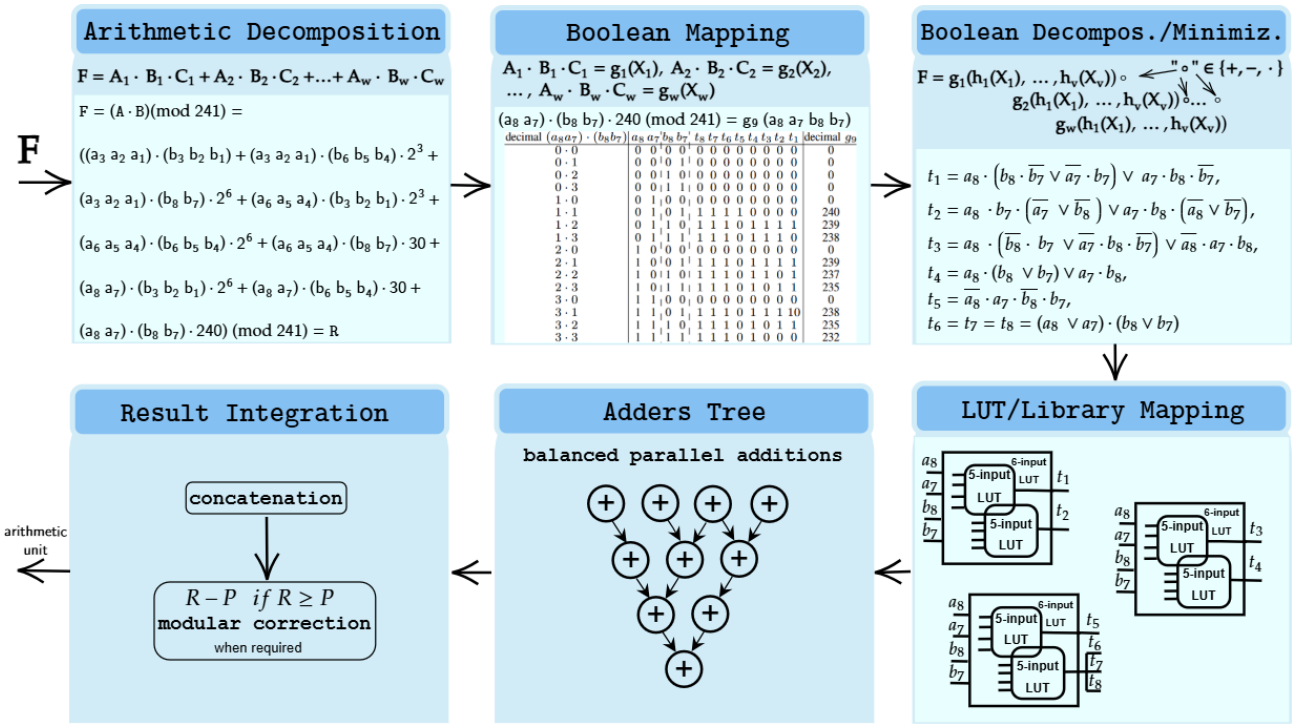

Fig. 1: Arithmetic function implementation methodology

applicable across different arithmetic operations while exploiting their unique characteristics. This methodology enhances delay and resource utilization compared to Xilinx Vivado and demonstrates competitive performance against FloPoCo by consistently delivering either more compact implementations or shorter critical paths, often achieving advantages in both metrics simultaneously. Comprehensive evaluation across ASIC technology (Cadence and Synopsys) confirms the approach's effectiveness across different implementation platforms.

The rest of the paper is organized as follows. Section II presents the six-stage methodology for arithmetic function decomposition. Section III provides experimental evaluation across all four arithmetic operations on FPGA and ASIC platforms. Section IV summarizes findings and discusses implications for arithmetic hardware design.

II. Methodology

The proposed methodology bridges the gap between high-level arithmetic specifications and efficient hardware implementations through a systematic six-stage pipeline, illustrated in Fig. 1. The figure also illustrates the evaluation in the first four stages for modular multiplication $R = (A \cdot B) (\mathrm{mod}\ 241)$ as a running example: an $8 \times 8 \rightarrow 8$ bit operation, where $A = (a_8 a_7 \ldots a_1)$, $B = (b_8 b_7 \ldots b_1)$, and $R = (r_8 r_7 \ldots r_1)$. The approach decomposes arithmetic operations into structured Boolean function networks specifically tailored for LUT-based architectures, achieving competitive results on both FPGAs and ASICs.

Stage 1: Arithmetic Decomposition. The input function F is decomposed into a superposition of small-bit multiplications:

$$\mathrm{F} = A_1 \cdot B_1 \cdot C_1 + A_2 \cdot B_2 \cdot C_2 + \cdots + A_w \cdot B_w \cdot C_w, \quad (1)$$

where F represents any of the arithmetic operations considered (e.g., modular reduction, modular multiplication, multiplication by a constant, or division by a constant), $A_1, A_2, \ldots, A_w, B_1, B_2, \ldots, B_w$ are m-bit binary sub-vectors derived from the input operands, and $C_1, C_2, \ldots, C_w$ are pre-calculated constants specific to the operation.

Stage 2: Boolean Mapping. Each multiplication term $A_i \cdot B_i \cdot C_i$ $(i = 1, 2, \ldots, w)$ is expressed as a system of Boolean functions $g_i(X_i)$, where X_i represents the concatenated input bits.

Stage 3: Boolean Decomposition and Minimization. The Boolean functions are decomposed into networks of subfunctions $h_j(X)$ with 5 or 6 variables each. Logic synthesis tools such as ABC [18] perform this decomposition, though some manual optimization is often required to fully exploit dual-output LUT capabilities [19].

Stage 4: LUT/Library Mapping. The decomposition strategy is inherently LUT-centric: 5-input subfunctions sharing variables are paired onto 6-LUTs to exploit dual-output capability. Notably, this same 6-input bounded decomposition proves equally effective both for FPGA and ASIC synthesis, structured small-function networks enabling synthesis tools to perform efficient technology mapping onto standard cells, a

benefit that emerges from the regularity of the decomposed structure rather than architecture-specific tuning.

Stage 5: Parallel Addition Network. LUT outputs feed a balanced addition tree performing parallel summation of intermediate results. Unlike one-by-one addition approaches [20], [21], this organization reduces critical path delay by computing higher-order and lower-order bits simultaneously. Moreover, when total input width fits within 6 variables, multiple operands (up to 3-4) can be combined in a single LUT, collapsing adder tree levels.

Stage 6: Result Integration. Final results are consolidated through concatenation of intermediate values. For modular operations, conditional correction ($R - P$ if $R \geq P$, where P is modulo) produces the final output.

This path is defined once per arithmetic operation class and is reused across all parameter values within that class (e.g., the same Stage 3 procedure applies to modular reduction for any modulus P). Adapting the methodology to a new operation class (rather than to specific operand values) is what makes the pipeline general and systematic.

In Fig. 1 for $(A \cdot B) \pmod{241}$, Stage 1 decomposes the operation into nine terms by splitting operands into 3- and 2-bit chunks, yielding functions of 4-, 5-, and 6-bit variables. Note that positional weights exceeding the modulus are reduced using modular equivalences ($2^9 \bmod 241 = 30$ and $2^{12} \bmod 241 = 240$):

$$\begin{aligned}
\big((a_6a_5a_4)\cdot(b_8b_7)\cdot 2^9\big)(\bmod\ 241) &= \\
\big((a_6a_5a_4)\cdot(b_8b_7)\cdot 30\big)(\bmod\ 241),& \\
\big((a_8a_7)\cdot(b_6b_5b_4)\cdot 2^9\big)(\bmod\ 241) &= \\
\big((a_8a_7)\cdot(b_6b_5b_4)\cdot 30\big)(\bmod\ 241),& \\
\big((a_8a_7)\cdot(b_8b_7)\cdot 2^{12}\big)(\bmod\ 241) &= \\
\big((a_8a_7)\cdot(b_8b_7)\cdot 240\big)(\bmod\ 241).&
\end{aligned} \tag{2}$$

Stage 2 represents each term as a Boolean function system: for example, the term $(a_8a_7)\cdot(b_8b_7)\cdot 240 \pmod{241}$ maps 4 input bits to 8 output bits via a truth table. Stage 3 minimizes these functions using ABC, producing expressions such as $t_6 = t_7 = t_8 = (a_8 \vee a_7)\cdot(b_8 \vee b_7)$. Stage 4 maps the optimized functions onto three 6-LUTs by pairing outputs that share identical input variables (t_1, t_2, t_3, t_4, and $t_5, t_6 = t_7 = t_8$) for the term $(a_8a_7)\cdot(b_8b_7)\cdot 240 \pmod{241}$.

III. Experimental Evaluation

We conducted experiments[1] with the following configurations: *lut_5* and *lut_6* represent 5-input and 6-input LUT decompositions, respectively; *Vivado* targets Xilinx embedded synthesis tool for FPGA; *Cadence* and *Synopsys* represent vendor synthesis tools for ASIC; *fpc* is the default setup of FloPoCo for multiplication by a constant, *fpc_0* and *fpc_3* correspond to FloPoCo solutions with `arch=0` (default architecture) and `arch=3` (architecture with arithmetic minimization), respectively, for 6-input LUTs (`alpha=6`). The work [20] considers similar conditions, with specific differences detailed in the division by constant analysis.

All approaches produce synthesizable Verilog RTL as output. For the proposed method the six-stage methodology generates hardware descriptions, in Verilog, where Boolean functions are explicitly decomposed according to the target LUT configuration. These hardware descriptions in Verilog files are then processed by the respective synthesis tools to obtain final metrics. FloPoCo similarly generates Verilog RTL, which undergoes identical synthesis flows. Native tool results (Vivado, Cadence, Synopsys) are obtained by synthesizing straightforward descriptions using standard arithmetic operators (e.g., `assign R = (A * B)% P;`), allowing the tools to apply their built-in optimization methods. The objective of this comparison is precisely to evaluate how the proposed methodology performs against the built-in optimization algorithms of EDA tools, with all tool-internal optimizations enabled. For elementary operations (e.g., 5x5 multipliers) native synthesis is known to be near-optimal, thus the research interest therefore lies in more complex arithmetic units, where the limits of generic synthesis algorithms become apparent.

FPGA experiments target Xilinx Kintex-7 (xc7k70tfbg484-3) using Vivado 2022.2 with complete place-and-route. To ensure fair comparison focusing on combinational logic optimization, the use of embedded hardware blocks (DSPs, BRAMs, hard multipliers) was disabled, Configurable Logic Blocks (CLBs), and their constituent LUTs, are the only FPGA components used. This constraint isolates the efficiency of Boolean function optimization from hardware resource specifics.

ASIC experiments employ two synthesis flows for identical technology: the Generic Standard Cell Library (gscl45nm) based on FreePDK45 for 45nm CMOS technology. **Cadence flow:** Logic synthesis via Cadence Genus 21.15-s080_1, physical implementation via Cadence Innovus 21.35-s114_1 . Results represent complete physical implementation. **Synopsys flow:** Logic synthesis via Synopsys Design Compiler U-2022.12. Results represent pre-layout synthesis-only metrics with wire load models for interconnect delay estimation. Both ASIC flows applied timing constraints of 1ns (aggressive optimization) for the presented results.

A. Multiplication by a Constant

Multiplication by a constant allows for significant optimizations since one operand is known at design time, enabling specialized circuit architectures that can achieve better resource utilization and improved performance.

Let's consider multipliers by a constant (C) using different five bit-ranges of A and five different constant values C, as it is specified $A \cdot C$ in Table I[2]:

1) 7-bit variable A and 29-bit prime $C = 536870909 = 2^{29} - 3$, as it titled $\mathbf{7 \times 29}$;
2) 8-bit variable A and 46-bit prime $C = 70368744177629$, as it titled $\mathbf{8 \times 46}$;

[1]HDL-files are in https://github.com/ZeboZebo702/DATE_2026.

[2]We use the symbol "·" to represent multiplication and the symbol "×" to the bit width of operands and values are represented in decimal.

3) 9-bit variable A and 101-bit prime constant $C = 25353\,012004564588029934064099 59$, as it titled $\mathbf{9 \times 101}$;
4) 9-bit variable A and 157-bit constant $C = 2^{157} - 7$, as it titled $\mathbf{9 \times 157}$;
5) 10-bit variable A and 183-bit composite constant $C = 2^{183} - 1$, as it titled $\mathbf{10 \times 183}$.

The results of the synthesis of the circuits for these five examples are presented in Table I. On FPGA, the proposed approach achieves advantages over FloPoCo and Vivado in LUT utilization ranging from 1.4×, for the 29-bit constant multiplication, to more than 11×, for the 183-bit special-form constant. Timing performance is approximately balanced across all three compared approaches with a slight advantage for the proposed method (up to 10% critical path improvement in some cases). The most substantial improvements are observed for special-form constants of the type $2^n \pm \psi$, for which Boolean minimization effectively recognizes repeating patterns in the binary representation.

TABLE I: Critical path and circuit area for the multiplication of A by a constant C

Vivado								
Multiplication	LUTs				critical path (ns)			
bits × bits	*Vivado*	*lut_5*	*lut_6*	*fpc*	*Vivado*	*lut_5*	*lut_6*	*fpc*
7× 29	24	**17**	**17**	24	8.0	7.4	**7.2**	7.9
8× 46	71	**46**	49	78	9	**8.1**	8.3	9.2
9 × 101	126	**79**	85	134	13.8	**12.2**	**12.2**	13.1
9 × 157	179	**44**	49	244	**14.7**	16.3	15.8	16.1
10 × 183	204	**18**	**18**	257	15.9	16.3	17	**15.5**
Cadence								
Multiplication	cells				critical path (ns)			
bits × bits	*Cadence*	*lut_5*	*lut_6*	*fpc*	*Cadence*	*lut_5*	*lut_6*	*fpc*
7× 29	**147**	228	161	161	0.8	**0.3**	0.4	0.6
8× 46	**306**	808	792	713	0.9	**0.7**	**0.7**	1.2
9 × 101	**387**	1063	1060	1660	**0.9**	**0.9**	**0.9**	1.7
9 × 157	**208**	483	464	964	0.8	0.6	**0.5**	1.4
10 × 183	138	**133**	**133**	795	**0.4**	0.5	0.5	1.2
Synopsys								
Multiplication	cells				critical path (ns)			
bits × bits	*Synopsys*	*lut_5*	*lut_6*	*fpc*	*Synopsys*	*lut_5*	*lut_6*	*fpc*
7× 29	2741	**298**	312	361	1.0	**0.4**	**0.4**	0.9
8× 46	4753	1074	**1051**	2502	1.2	0.9	**0.7**	1.1
9 × 101	10804	1620	**1601**	5798	1.3	**0.9**	1.0	1.5
9 × 157	17816	617	**599**	8184	1.4	0.6	**0.5**	1.3
10 × 183	23348	246	**238**	8576	1.4	**0.4**	0.5	1.4

Synthesis in Cadence yields a more complex picture. Native Cadence algorithms demonstrated greater area efficiency for most cases, utilizing 1.5-2.7x fewer cells than the proposed approach for general-form constants. The exception is the 183-bit special-form constant, where the proposed method achieves comparable cell count (133 vs 138). Compared to FloPoCo, the proposed approach uses comparable or fewer cells (up to 6x fewer depending on the constant). However, timing performance of the proposed approach is more favorable for most configurations, achieving critical paths of 0.4-0.9ns versus 0.6-1.7ns for FloPoCo. Compared to native Cadence, the proposed method achieves up to 2x advantage for general-form constants (e.g., 0.4ns vs 0.8ns for 7x29), though for the 183-bit special-form constant Cadence achieves a shorter critical path (0.4ns vs 0.5ns). These results demonstrate the ability of Boolean decomposition to create shorter critical paths through parallelized computational structures.

For Synopsys synthesis, the area advantage varies from 9× for the 29-bit constant multiplication to nearly 100× for the 183-bit special-form constant compared to native Synopsys synthesis. The advantage over FloPoCo ranges from 1.2× to 36× depending on the constant value. The critical path of the proposed approach is up to 3.5× shorter than native Synopsys and up to 2.8× shorter than FloPoCo.

B. Modular Multiplication

Modular multiplication computes $(A \cdot B)(\mathrm{mod}\ P)$, where A and B are input operands and P is the modulo. We focus on modular multiplication for small moduli, which form the building blocks of RNS computations; the efficiency of the overall system depends critically on the efficiency of these elemental operations. The foundation of RNS relies on computations with moduli that must be co-prime numbers. Hardware implementation of modular multiplication for small moduli enables high parallelism in RNS-based systems. Some efficient approaches to RNS computations involves operations with small bit-width moduli ranging from 4 to 12 bits [22], [23].

The propose methodology applies Boolean function minimization techniques to represent the modular multiplication result directly as a function of the input bits, bypassing the need for explicit multiplication followed by reduction. This approach is particularly effective for small moduli, where the complete truth table of the operation can be represented and minimized efficiently. Efficient mapping of modular multiplication is based on (3).

$$(A \cdot B)(\mathrm{mod}\ P) = \sum_{i=1}^{w}\sum_{j=1}^{w}\left(A_i \cdot B_j \cdot 2^{(i+j-2)\cdot\delta}\right)(\mathrm{mod}\ P) = S, \tag{3}$$

where $A = (A_w, A_{w-1}, \ldots, A_1)$ and $B = (B_w, B_{w-1}, \ldots, B_1)$ are w-th bit sub-vectors, δ the number of bits per sub-vector, and A and B are n-bit vectors. One tricky aspect of the proposed approach is the selection of the appropriate value for 2δ. Specifically, with k denoting the number of input bits of the LUTs, when $i = j = w$, we require that $2\delta \leq k$, while for cases where $i, j \neq w$, we set $2 \cdot \delta = k$.

Table II presents experimental results for modular multipliers with five different prime moduli: $P = 241, 491, 997, 2011, 4051$, covering 8 to 12-bit ranges. It is important to note that FloPoCo does not support modular multiplier generation, therefore comparison is limited to the proposed approach and vendor synthesis tools.

On FPGA, the proposed approach demonstrates modulus-size-dependent optimization characteristics. For small moduli ($P = 241, 491$), the proposed method achieves balanced improvements with up to 25% LUT reduction and up to 25% critical path reduction compared to Vivado. For larger

moduli ($P = 997, 2011, 4051$), the optimization shifts toward timing-focused strategy, showing up to 30% LUT increase but delivering consistent up to 15% delay improvements. This evolution reflects the trade-off between compact Boolean function representations for smaller moduli, and parallelized structures for improved timing in larger moduli.

TABLE II: Critical path and circuit area for modular multipliers $(A \cdot B)(\mathrm{mod} P)$

Vivado						
P	LUTs			critical path (ns)		
	Vivado	*lut_5*	*lut_6*	*Vivado*	*lut_5*	*lut_6*
241	145	128	**127**	13.6	10.9	**10.7**
491	162	**125**	129	13.9	**10.5**	10.9
997	**232**	244	245	14.8	**13.8**	13.9
2011	**219**	282	277	16.2	14.0	**14.3**
4051	298	**295**	310	17.1	**14.4**	14.9
Cadence						
P	cells			critical path (ns)		
	Cadence	*lut_5*	*lut_6*	*Cadence*	*lut_5*	*lut_6*
241	**2083**	2279	2199	2.7	**2.0**	2.1
491	**2335**	2852	2891	3.1	**2.2**	**2.2**
997	**3288**	3624	3512	4.0	**2.6**	**2.6**
2011	**3613**	4225	4094	4.6	2.6	**2.3**
4051	**4518**	5322	5355	5.2	**2.8**	3.0
Synopsys						
P	cells			critical path (ns)		
	Synopsys	*lut_5*	*lut_6*	*Synopsys*	*lut_5*	*lut_6*
241	**2653**	4409	4276	3.0	**1.5**	**1.5**
491	**3302**	5395	5103	3.5	**1.5**	**1.5**
997	**3912**	7739	7156	3.8	**1.7**	1.8
2011	**4705**	8603	8131	4.4	**1.7**	**1.7**
4051	**5748**	10447	9875	4.7	**1.6**	**1.6**

Synthesis in Cadence reveals consistent timing advantages of the proposed approach despite variable area characteristics. For small moduli ($P = 241$), the proposed approach uses comparable cell count while achieving 20% delay improvement (2.1ns vs 2.7ns). For larger moduli ($P = 2011, 4051$), the area overhead increases to up to 20%, but timing improvements become dramatic up to 50% shorter critical paths (2.3-3.0ns vs 4.6-5.2ns). This pattern demonstrates that Boolean decomposition creates fundamentally shorter datapaths through parallelized multiplication of small-bitwidth factors, which becomes particularly effective during physical implementation for larger moduli.

For Synopsys synthesis, the proposed approach utilizes 1.6-1.8× more cells than native Synopsys across all moduli sizes. However, regarding performance, significant shorter critical paths are achieved: 1.5-1.8ns for the proposed method versus 3.0-4.7ns for native Synopsys, representing up to 60% timing improvement. The proposed method's delays show minimal dependence on modulus size (1.5-1.8ns range), while native Synopsys delays increase with modulus (3.0-4.7ns), indicating more efficient critical path organization in the decomposition-based approach.

C. Modular Reduction

Modular reduction $A \pmod P$ is represented as the n-bit number $A = (A_w, A_{w-1}, \ldots, A_1)$ split into w 6-bit chunks, multiplying the i^{th} chunk by $2^i (\mathrm{mod}\ P)$ for $i = 1, 2, \ldots, w$. Splitting into 6-bit chunks reduces the number of intermediate additions compared with splitting into 5-bit chunks. The efficient mapping of the reduction is based on (4).

$$A(\mathrm{mod}\ P) = \sum_{i=1}^{w} A_i \cdot \left(2^{6\cdot(i-1)} (\mathrm{mod}\ P)\right). \tag{4}$$

Table III presents experimental results for modular reduction with two input bitwidths (168-bit and 270-bit) and five different moduli ($P = 241, 491, 997, 2011, 4051$). On FPGA, the proposed approach demonstrates the most significant improvements among all operations. For 168-bit inputs, LUT reduction ranges up to 10x (400-599 vs 4290-4706 for Vivado) with simultaneous up to 35% critical path reduction. For 270-bit inputs, the advantages are even more pronounced: up to 20x LUT reduction (509-834 vs 9926-11670), with the 241-modulo case achieving a remarkable 20× reduction (509 vs 10024 LUTs) and 40% critical path reduction. This exceptional efficiency stems from the decomposition into weighted chunk multiplications $\sum_i A_i \cdot (2^{6(i-1)} \bmod P)$, where each chunk-constant product represents a small Boolean function. Compared to FloPoCo (*fpc_3*), the proposed approach in eight cases of ten achieves up to 20% LUT reduction, though *fpc_3* achieves up to 25% better delays in some cases.

Synthesis in Cadence reveals the a complex optimization landscape. Native Cadence synthesis shows anomalously large delays (40–280 ns for 168–270-bit inputs), orders of magnitude worse than other operations, indicating severe routing congestion or non-optimized datapath structures. The proposed approach achieves up to 20× cell reduction with huge timing improvements: up to 85× better critical paths (for 270-bit inputs with $P = 491$). Compared to FloPoCo, the proposed method uses up to 1.8× fewer cells than *fpc_0* and *fpc_3*, reaching 30% outperform in the critical path. Notably, *fpc_0* often produces more compact circuits than *fpc_3* on Cadence, yet the proposed approach consistently outperforms both variants.

For Synopsys synthesis, the proposed approach achieves up to 2.3× cell count reduction compared to native Synopsys. More critically, timing improvements are significant: up to 30× better critical paths (2.0–2.6 ns vs 33.8–68.2 ns). All approaches (*lut_6*, *fpc_0*, *fpc_3*) show nearly uniform delays (2.0–2.6 ns) on Synopsys, characteristic of wire load model limitations in distinguishing subtle datapath topology differences. However, area differences are substantial: the proposed method uses 2–3× fewer cells than *fpc_0* and 2.7–5.3× fewer cells than *fpc_3*, with the advantage increasing for mid-range moduli ($P = 491, 997, 2011$). Native Synopsys exhibits extremely long delays, indicating fundamental inability to efficiently synthesize modular reduction without specialized algorithms.

D. Division by a Constant

The proposed approach for division by a constant entirely avoids memory elements and feedback loops. The dividend is decomposed into a network of small-bit numbers, which are

represented as systems of Boolean functions. These systems of functions encode the quotient and remainder calculations for their respective sub-vectors, specifically tailored to LUT architectures. The architecture's scalability stems from its ability to handle divisors of arbitrary value and bit-width using a systematic decomposition approach, rather than relying on special-case algorithms restricted to specific types of divisors. The proposed method is particularly valuable for applications where the divisor is known at design time, allowing for specialized optimizations that would be not possible in general-purpose dividers.

TABLE III: Critical path and circuit area for modular reduction $A(\mathrm{mod}\ P)$

Vivado									
bit range of A	P	LUTs				critical path (ns)			
		Vivado	*lut_6*	*fpc_0*	*fpc_3*	*Vivado*	*lut_6*	*fpc_0*	*fpc_3*
168	241	4419	400	7979	**380**	25.7	17.2	238.4	**16.1**
	491	4330	**467**	12333	508	23.6	18.4	281.4	**17.9**
	997	4706	599	13198	**558**	23.6	17.0	298.2	**16.6**
	2011	4290	**534**	14847	610	25.9	17.7	308.8	**16.5**
	4051	4225	**484**	4691	529	25.3	17.5	263.4	**17.3**
270	241	10024	**509**	13847	604	33.6	20.5	382.6	**17.2**
	491	11088	**701**	17185	826	33.7	23.1	445.7	**17.3**
	997	11310	**760**	15061	905	30.7	22.4	444.9	**17.2**
	2011	11670	**834**	10454	1005	34.2	21.9	415.0	**18.0**
	4051	9926	**744**	7751	861	33.4	21.5	426.8	**18.6**
Cadence									
bit range of A	P	cells				critical path (ns)			
		Cadence	*lut_6*	*fpc_0*	*fpc_3*	*Cadence*	*lut_6*	*fpc_0*	*fpc_3*
168	241	19474	**5274**	9447	8417	40.2	**2.3**	2.7	2.7
	491	21041	**13299**	14722	16519	49.1	**2.9**	3.2	3.1
	997	31824	**14280**	14868	16519	58.7	**2.8**	2.9	3.1
	2011	33253	**16388**	17588	28828	60.7	**2.5**	3.4	3.4
	4051	27766	**10840**	13782	14095	51.3	**2.7**	3.3	3.0
270	241	164957	**8178**	14489	14134	210.0	**2.6**	3.0	2.8
	491	225954	**21381**	24630	27730	280.0	**3.3**	3.9	3.7
	997	213834	**21801**	26337	29756	268.0	**3.2**	3.7	3.9
	2011	173843	**25866**	29421	34086	198.0	**3.3**	4.0	4.1
	4051	197444	**16374**	22502	24480	248.3	**2.9**	4.0	4.3
Synopsys									
bit range of A	P	cells				critical path (ns)			
		Synopsys	*lut_6*	*fpc_0*	*fpc_3*	*Synopsys*	*lut_6*	*fpc_0*	*fpc_3*
168	241	23155	**10486**	19488	27827	33.8	2.13	**2.0**	**2.0**
	491	26315	**19275**	47380	69012	34.3	2.3	**2.1**	2.2
	997	28844	**18799**	49922	68247	35.1	**2.1**	2.2	2.3
	2011	32367	**22717**	44571	70130	37.3	**2.1**	2.3	2.3
	4051	35757	**16266**	30253	40081	38.4	**2.0**	2.2	2.2
270	241	36428	**15758**	32067	42038	56.4	**2.3**	**2.3**	**2.3**
	491	42121	**28570**	89559	150875	57.6	2.6	**2.5**	**2.5**
	997	44700	**29113**	89706	143962	60.1	2.5	**2.4**	2.5
	2011	49225	**33825**	88314	139098	64.8	**2.3**	2.5	2.6
	4051	52826	**24686**	51673	65493	68.2	**2.2**	2.5	2.5

Let's consider the division of the dividend A by a d, the quotient is Q and the residue R, as expressed in (5), where A, d, Q, and R are integers.

$$\begin{aligned} A = Q \cdot d + R\ ; \qquad & \frac{A}{d} \equiv \{Q, R\}; \\ \left\lfloor \frac{A}{d} \right\rfloor = Q; \qquad & R = A - Q \cdot d. \end{aligned} \tag{5}$$

The proposed division method is similar to the one adopted for modular reduction: the dividend is split into 6-bit chunks starting from the first v-bit segment, where v is the bit width of the divisor. Hence, A is considered as a sequence of w 6-bit chunks, with $i = 2, 3, \ldots w$; chunk A_1 is represented by a v-bit chunk, as specified in (6).

$$\begin{aligned} R_1 = \begin{cases} A_1 - d, & \text{if } d \le A_1; \\ A_1, & \text{otherwise;} \end{cases} \quad Q_1 = \begin{cases} 1, & \text{if } d \le A_1; \\ 0, & \text{otherwise;} \end{cases} \\ 2^{6\cdot(i-2)+v} \cdot A_i = & Q_i \cdot d + R_i; \\ \frac{2^{6\cdot(i-2)+v} \cdot A_i}{d} \equiv & \{Q_i, R_i\}; \\ \left\lfloor \frac{2^{6\cdot(i-2)+v} \cdot A_i}{d} \right\rfloor = & Q_i; \\ 2^{6\cdot(i-2)+v} \cdot A_i - Q_i \cdot d = & R_i. \end{aligned} \tag{6}$$

From (5), the division can be represented as the sum of (6):

$$Q_{temp} = Q_1 + \sum_{i=2}^{w} Q_i; \qquad R_{temp} = R_1 + \sum_{i=2}^{w} R_i,$$

where Q_1 and R_1 are the quotient and the residue of the division of the first v-bit chunk, respectively. The final result $\{Q, R\}$ is represented as:

$$Q = Q_{temp} + \left\lfloor \frac{R_t}{d} \right\rfloor; \quad R = R_{temp} - \left\lfloor \frac{R_{temp}}{d} \right\rfloor \cdot d.$$

Table IV presents experimental results for division by constant with four dividend bitwidths (16, 32, 48, 64 bits) and various divisor values. With the purpose of comparison, it provides also results for the approach in [20] and FloPoCo variants.

On FPGA, the proposed approach exhibits strong input-bitwidth-dependent optimization with a significant crossover at 32 bits. For 16-bit dividends, Vivado maintains area advantage (30-37 LUTs vs 64-66 for *lut_6*), while the proposed method achieves up to 20% better timing. At 32-bit dividends, a radical shift occurs: the proposed approach achieves up to 6x LUT reduction with up to 40% delay reduction. This crossover reflects Vivado's effective pattern matching for small dividends versus the proposed method's systematic 6-bit chunk decomposition that scales linearly. For 48-64-bit dividends, advantages of the proposed methodology strengthen to up to 5x LUT savings with up to 1.7x timing improvements. Compared to [20], the proposed method shows comparable performance for 16-32-bit cases. For 48-bit dividends, the proposed approach achieves up to 25% fewer LUTs and up to 1.7× shorter critical path. For 64-bit dividends, [20] achieves fewer LUTs in all cases, while the proposed method delivers up to 2.5× better critical paths. Against FloPoCo, the proposed approach consistently achieves better critical path and comparable area costs.

Synthesis with Cadence tools demonstrates systematic performance advantage across the entire input range. For 16-bit dividends, the proposed approach shows mixed results: +50% cells with +20% delay for divisor 5, but -40% cells with -35% delay for divisor 13. For 32-bit dividends, timing advantages become dominant: -10% to +18% area variation, but up to

3.6x delay improvements, with the divisor 23 case achieving simultaneous benefits (-12% cells, -72% delay). For 48-bit dividends, the pattern strengthens: up to +15% area overhead but dramatic up to 74% timing reduction (2.1-2.3ns vs 9.1-9.7ns), representing 4-4.5× shorter critical paths. For 64-bit dividends, area overhead increases to up to 55% but reduction achieves 4.5-4.7× (2.2-2.4ns vs 9.8-11.2ns). Compared to [20], the proposed method outperforms for 32, 48, and 64-bit dividends, with advantages strengthening for larger bitwidths: up to 25% fewer cells and up to 2.2x better timing for 48-64-bit cases. Against FloPoCo, *lut_6* uses 0.8-1.6× cells of *fpc_3* while consistently achieving 60% better timing, demonstrating that Boolean decomposition creates shorter critical paths through parallelized computation structures, at the cost of additional logic resources for larger dividends.

For Synopsys synthesis, the proposed approach demonstrates consistent and substantial timing advantages across all dividend bitwidths, with area costs that varies with operand size. For 16-bit dividends, *lut_6* achieves comparable or less area in two of three cases (up to 1.5× fewer cells for $d = 13$) with 2× timing improvement (1.0–1.1 ns vs 2.2–2.3 ns). For 32-bit dividends, area overhead increases moderately (1.1–1.4× more cells), but timing reduces to 4.7× (1.1–1.2 ns vs 4.2–5.2 ns). For 48-bit dividends, the pattern shows mixed area results: *lut_6* uses up to 1.3× more cells for smaller divisors but achieves 1.2× fewer cells for $d = 241$, while timing improvements reach 7–7.5× (1.1–1.2 ns vs 8.3–8.5 ns). For 64-bit dividends, area overhead is most pronounced (1.6–2.7× more cells), yet timing advantages peak at 8–9× (1.1–1.2 ns vs 8.8–11.1 ns). The near-uniform delays (1.0–1.2 ns) across all bitwidths for the proposed approach contrast sharply with native Synopsys delays that grow from 2.2 ns to 11.1 ns, demonstrating fundamentally more efficient critical path organization through parallelized Boolean decomposition. Compared to FloPoCo *fpc_3*, *lut_6* uses 1.1–2.4× fewer cells while achieving 1.3–1.8× better timing across all configurations. Compared to [20], the proposed method achieves nearly identical cell counts with a slight advantage, while timing improvements scale with dividend bitwidth: comparable delays for 16-bit dividends, increasing to 4× shorter critical paths for 64-bit dividends.

TABLE IV: Critical path and circuit area for division by a constant $\frac{A}{d} \equiv \{Q, R\}$

Vivado											
bit range of A	d	LUTs					critical path (ns)				
		Vivado	*[20]*	*lut_6*	*fpc_0*	*fpc_3*	*Vivado*	*[20]*	*lut_6*	*fpc_0*	*fpc_3*
16	5	**30**	51	64	71	59	8.0	8.7	**7.7**	8.4	9.0
	11	**37**	52	57	154	59	10.6	**8.5**	8.6	9.7	9.6
	13	**37**	70	66	184	63	10.3	8.7	**8.1**	9.2	9.0
32	5	718	125	**123**	158	155	15.6	15.8	**9.5**	12.9	10.7
	11	563	**158**	165	370	182	15.8	16.4	**10.1**	14.1	11.5
	23	445	**186**	187	979	197	14.7	15.3	**11.1**	18.7	11.7
48	47	1144	**394**	423	2849	410	18.4	22.8	**13.5**	29.5	13.7
	113	1382	501	523	4672	**408**	17.6	19.7	**13.4**	38.1	14.3
	241	1596	750	575	4713	**386**	18.4	17.2	**13.4**	35.5	14.3
64	5	2302	385	460	**341**	523	21.1	30.4	**12.8**	22.0	14.7
	11	2039	**435**	467	955	589	22.1	32.3	**12.5**	27.0	16.0
	23	1532	**493**	514	2454	542	19.2	33.0	15.5	35.7	**15.3**
Cadence											
bit range of A	d	cells					critical path (ns)				
		Cadence	*[20]*	*lut_6*	*fpc_0*	*fpc_3*	*Cadence*	*[20]*	*lut_6*	*fpc_0*	*fpc_3*
16	5	**762**	1281	1131	1110	963	**1.1**	1.3	1.3	1.3	1.4
	11	**1335**	1413	1344	3225	1346	1.9	**1.4**	**1.4**	2.2	1.7
	13	1403	1097	**844**	4176	1409	1.8	**1.2**	**1.2**	3.2	1.4
32	5	**2197**	3371	2564	2347	2741	**1.3**	2.1	1.4	2.9	2.0
	11	**3108**	4132	3677	6774	3847	4.4	2.1	**1.7**	5.4	2.3
	23	3999	3840	**3520**	12277	4053	5.0	2.0	**1.4**	7.0	2.2
48	47	**7284**	9806	8336	29449	9781	9.1	2.9	**2.1**	21.1	3.0
	113	8134	8616	8288	47828	**8106**	9.6	3.0	**2.3**	31.2	2.8
	241	9500	7401	7393	36713	**6345**	9.7	2.5	**2.2**	20.7	2.8
64	5	5485	10193	7662	**4825**	9768	2.7	4.2	**1.9**	6.2	3.2
	11	**6033**	11311	9361	15324	12553	9.8	3.9	**2.2**	14.6	3.5
	23	**7903**	10467	9788	27861	11804	11.2	3.5	**2.4**	16.8	3.3
Synopsys											
bit range of A	d	cells					critical path (ns)				
		Synopsys	*[20]*	*lut_6*	*fpc_0*	*fpc_3*	*Synopsys*	*[20]*	*lut_6*	*fpc_0*	*fpc_3*
16	5	**1445**	2110	1589	3199	2509	2.2	1.1	**1.0**	1.1	1.3
	11	1945	2221	**1939**	9980	2701	2.3	**1.1**	**1.1**	**1.1**	1.5
	13	1929	1927	**1250**	10861	3010	2.2	1.1	**1.0**	1.1	1.4
32	5	**3040**	5003	4189	7752	5768	4.2	2.2	**1.2**	1.5	1.5
	11	**4097**	5970	5430	21508	6503	5.0	2.1	**1.1**	2.2	1.6
	23	**4990**	6090	5398	36635	7138	5.2	2.1	**1.1**	2.4	1.7
48	47	**9414**	12709	12620	80152	15172	8.3	3.0	**1.1**	3.8	1.9
	113	**10315**	12212	12038	153050	14748	8.3	2.6	**1.1**	4.5	2.1
	241	11982	**10021**	**10021**	336409	11725	8.5	2.2	**1.2**	5.1	2.0
64	5	**5883**	15615	16068	14035	16030	8.8	4.5	**1.1**	2.9	2.0
	11	**8269**	17185	17121	38097	17102	9.9	4.3	**1.2**	4.4	2.1
	23	**10251**	15780	16090	62043	18741	11.1	4.0	**1.2**	4.8	2.0

IV. Discussion and Conclusion

This paper proposed a systematic methodology for designing efficient arithmetic units on FPGA and ASIC platforms through Boolean function decomposition tailored specifically for LUT architectures. The six-stage approach demonstrates consistent performance improvements across diverse arithmetic operations (multiplication by constant, modular multiplication, modular reduction, and division by constant).

Experimental results show that the methodology achieves competitive or superior performance compared to vendor synthesis tools and FloPoCo framework across both FPGA and ASIC platforms. On FPGA, the approach consistently provides either significant resource savings (up to 20× LUT reduction for modular reduction) or substantial timing improvements (up to 45% critical path reduction for division), often achieving advantages in both metrics simultaneously. ASIC synthesis reveals platform-specific characteristics: Synopsys pre-layout results demonstrate dramatic cell count reductions (up to 100× for multiplication by constant) with consistent timing advantages, while Cadence post-layout synthesis shows more nuanced trade-offs, exceptional timing performance (up to 85× for modular reduction) with area efficiency varying by operation type.

Particularly noteworthy is the method's effectiveness for modular reduction, addressing a fundamental weakness in traditional synthesis tools that lack specialized algorithms for this operation. Native tools produce inefficient implementations: Cadence yields up to 85× longer critical paths and up to 28×

more cells, while Synopsys exhibits up to 30× longer critical paths and up to 2.3× more cells.

The paper presented ASIC results for 1ns timing constraints to evaluate aggressive optimization scenarios. Additional experiments conducted across unconstrained (area-optimal) and 10ns (relaxed timing) constraints reveal consistent trends. By relaxing timing constraints from 1ns to unconstrained typically yields 5-20% area reduction with increasing critical path (sometimes up to 4x). Comprehensive power analysis (across multiple standard cell libraries and composite metrics) constitutes a primary direction for future work, motivated by preliminary measurements on gscl45nm that do not show a clear correlation between power and either delay or area.

Despite being formulated for FPGA LUT architectures, the proposed approach demonstrates effectiveness also on ASIC platforms. The key insight is that decomposing arithmetic operations into bounded 6-input Boolean function networks creates a regularity that benefits any synthesis flow. For FPGAs, this structure directly maps to LUT primitives, while for ASICs it provides synthesis tools with manageable subproblems rather than monolithic Boolean networks spanning dozens or hundreds of inputs. Standard cell mappers can efficiently optimize small, well-defined functions, whereas large networks often cause synthesis tools to expend excessive runtime searching for optimal decompositions of large Boolean networks. The resulting structured netlists exhibit shorter critical paths through inherent parallelism and reduced logic depth benefits that manifest regardless of whether the target is an FPGA LUT or an ASIC standard cell. This structural regularity, rather than LUT specific targeting, explains the cross-platform effectiveness of the approach.

Synthesis runtime represents another practical advantage. The proposed approach requires at most several minutes for the most complex designs (e.g., 270-bit modular reduction). In contrast, vendor synthesis tools and FloPoCo can require many hours for equivalent designs, particularly for operations like modular reduction where specialized optimization algorithms must explore large solution spaces.

The strength of the proposed methodology lies in its adaptability to the specific characteristics of different arithmetic operations, rather than relying on one-size-fits-all optimization strategies. While the full six-stage strategy is illustrated for modular multiplication, individual operations require only specific subsets of stages: multiplication by a constant uses only Stages 2-4 (Boolean mapping through LUT mapping), division by a constant omits modular correction, and modular multiplication requires correction but not concatenation. This modularity suggests broader applicability beyond the four operations studied here, including floating-point arithmetic and matrix operations, which represents a direction for future work. This flexibility is particularly valuable for RNS implementations, where traditional approaches often fail to deliver efficient solutions.

ACKNOWLEDGMENT

This work was partially supported by Portuguese national funds through Fundação para a Ciência e a Tecnologia I.P. (FCT) under projects UID/50021/2025 and UID/PRR/50021/2025, and European Defence Fund (EDF) grant agreement No. 101168112 (SEQURED).

REFERENCES

[1] F. de Dinechin, S.-I. Filip, L. Forget, and Martin Kumm, "Table-Based versus Shift-And-Add constant multipliers for FPGAs", 26th IEEE Symposium on Computer Arithmetic (ARITH '19), 2019, Kyoto, Japan. pp.1-8.

[2] E.G. Walters III, "Reduced-Area Constant-Coefficient and Multiple-Constant Multipliers for Xilinx FPGAs with 6-Input LUTs", Electronics No. 6, 2017, pp. 1-29.

[3] P.V.A. Mohan, "Residue Number System. Theory and applications", Springer International Publishing, 2016.

[4] A.R. Omondi, "Cryptography Arithmetic. Algorithms and Hardware Architectures", Springer Nature Switzerland, 2020.

[5] L. Sousa, S. Antao, P. Martins, "Combining Residue Arithmetic to Design Efficient Cryptographic Circuits and Systems", IEEE Circuits and Systems Magazine, Vol. 16, N 4, 2016, pp. 6-32.

[6] H. Flatt, S. Hesselbarth, S. Flugel, and P. Pirsch, "A Modular Coprocessor Architecture for Embedded Real-Time Image and Video Signal Processing", Embedded Computer Systems: Architectures, Modeling, and Simulation, 7th International Workshop, 2007, Samos, Greece, Proceedings, p. 241-250.

[7] 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.

[8] D. Harris, S. Harris, "Digital Design and Computer Architecture", Morgan Kaufmann publishers; 2nd edition, 2012, 720 p.

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

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

[11] 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.

[12] 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.

[13] 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.

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

[15] https://flopoco.org/

[16] F. de Dinechin, B. Pasca, "Designing custom arithmetic data paths with FloPoCo", IEEE Design & Test of Computers, 28(4), 18–27, July 2011.

[17] F. de Dinechin, M. Kumm, "Application-Specific Arithmetic", Springer, 2024.

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

[19] D. Gorodecky, "Two approaches of arithmetic units design", it - Information Technology (De Gruyter), Vol. 66, Issue 4-5, 2025, pp-114-123.

[20] D. Gorodecky and L. Sousa, " Scalable architecture of constant division on FPGA", Proceedings of the 30th IEEE Symposium on Computer Arithmetic, Sep. 4-6, 2023, Portland, Oregon, USA.

[21] D. Gorodecky, L. Sousa, "Modular Arithmetic Based on Boolean Functions: A Divide and Conquer Approach," in IEEE Access, vol. 13, pp. 186383-186396, 2025, doi: 10.1109/ACCESS.2025.3626755.

[22] J. Y. S. Low and C.-H. Chang, "A New Approach to the Design of Efficient Residue Generators for Arbitrary Moduli," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 60, no. 9, pp. 2366-2374, Sept. 2013, doi: 10.1109/TCSI.2013.2246211

[23] H. Nakahara and T. Sasao, "A High-speed Low-power Deep Neural Network on an FPGA based on the Nested RNS: Applied to an Object Detector," 2018 IEEE International Symposium on Circuits and Systems (ISCAS), Florence, Italy, 2018, pp. 1-5, doi: 10.1109/ISCAS.2018.8351850.