Constant Multiplication Based on Boolean Minimization

Danila Gorodecky and Petr Bibilo

National Academy of Science of Belarus, Minsk, Belarus
danila.gorodecky@gmail.com
bibilo@newman.bas-net.by

Abstract. This contribution studies constant multiplication $X \cdot C$ and $X \cdot C \ (mod\ P)$, where constant C achieves 2263 bits and variable X varies up to 10 bits. We aim to develop an efficient approach to design constant multiplication for FPGA and ASIC. The proposed technique demonstrates up to 2 times advantage in performance and up to 5 times advantage in area costs under direct multiplication on FPGA, up to 7 times in performance in custom library on ASIC.

Keywords: Constant multiplication · modular multiplication · multiplier · Boolean minimization · BDD · ROBDD · computer arithmetic · RNS · FPGA · ASIC.

1 Introduction

The constant multiplication (CM) is an important operation [1–13] that appears, for example, in many numerical algorithms, signal, image, video processing applications, cryptography, neural networks, and the residue number system (RNS).

Multiplication has a specific realization depending on implementation on application-specific integrated circuit (ASIC) and field-programmable gate array (FPGA).

The efficiency of multipliers on ASIC is strongly related to the libraries elements and can significantly vary depending on the library.

Contemporary FPGAs have embedded hard multipliers distributed throughout the fabric due to the importance of multiplication. Nonetheless multiplication based on lookup tables (LUTs) in the configurable logic fabric are often used for high-performance designs for several reasons [11]:

- embedded multiplier operands are fixed in size and type, such as 25×18 two's complement, while LUT-based multiplication can be any size or type;
- the number and location of embedded multipliers are fixed, while LUT-based multiplication can be placed anywhere and the number is limited only by the size of the reconfigurable fabric;
- embedded multipliers cannot be modified, while LUT-based multiplication can use techniques such as merged arithmetic [14], approximate arithmetic [15], and multiplication by modulo [12, 16] to optimize the overall system.

One common and efficient way for realizing CM (and multiplication in common) circuits is to transform them into several additions, subtractions and bit-shifts [1–11]. On higher level of abstraction can be implemented wide spread in such algorithms as schoolbook multiplication [17], Karatsuba multiplication [18], Fourier multiplication [19], Booth multiplication [20], Montgomery multiplication by modulo [21] and many others [17, 22].

Number of unities in binary representation can be considered as significant feature for the efficient multiplication on deeper level of abstraction[6, 4].

We propose a technique of CM design based on Boolean minimization focusing on further implementation in multioperand addition in neural networks and backward transformation in RNS. This technique of CM design is suitable for standard multiplication $X \cdot C$ and multiplication by modulo $X \cdot C$ $(modP)$ for up to thousands bits of constant C and modulo P, but it is limited up to 14-16 bits of input variable X.

Following this brief introduction, Section 2 explains Boolean functions implementation in the context of CM design and considers examples, Section 3 demonstrates result of the synthesis on FPGA and ASIC, compares critical paths and area costs with direct multiplication proposed by Xilinx and Mentor Graphics, finally we summarize the results and plan further research in Section 4.

2 Representation of Constant Multiplication by Boolean Functions

The result of any arithmetic computation can be represented or approximated by sum-of-products (SOPs). However the original representation given by truth tables may be unmanageable by synthesis tools, e.g., the truth table of the product of two 16-bit input operands requires 64 columns (16 columns for each operand and 32 columns for the result) and more than four billions rows.

Nevertheless truth table representation might be efficiently implemented in the design of combinational logic. We figured out that representation of arithmetic operations by up to 2^{12} rows truth table with further two-level minimization of Boolean functions [23, 25] is efficient [26, 27]. The performance and the area costs of arithmetic units designed with two-level minimization comparing with standard electronic design automation (EDA) tools achieve 30 times.

This contribution considers CM by using multi-level minimization based on:

- truth table representation of the results of arithmetic functions;
- binary decision diagram (BDD) minimization with parsing by j subfunctions.

2.1 Truth Table Representation of Constant Multiplication

In the context of arithmetic operations (i.e. $X + C = R$, $X \cdot C = R$, $X \cdot C$ $(mod\ P) = R$ and etc.) the constant C can be represented as the truth table of a system of Boolean functions $r_n, r_{n-1}, \ldots, r_1$ depended on variable binary inputs $x_m, x_{m-1}, \ldots, x_1$ without dependence on binary representation of a

constant $c_k, c_{k-1}, \ldots, c_1$, where $R = (r_n, r_{n-1}, \ldots, r_1)$, $X = (x_m, x_{m-1}, \ldots, x_1)$, and $C = (c_k, c_{k-1}, \ldots, c_1)$.

Implementation of truth table allows to represent any arithmetic operation under a really big value of constants (in thousand bit-ranges), but limited by small bit-range of input variables (up to a couple of dozen of bits). Two-level minimization of the system of Boolean functions imposes additional restriction on bit-range of input variable X and limited to $15 - 20$ variables depending on number of Boolean function n. For example, two-level minimizer Espresso is corrupted of truth table for n,$m > 15$. This question has been investigated in [26, 27].

We propose to represent arithmetic operations in small bit-range of inputs and up to thousands of bits of constant. This concept is easily demonstrated on the truth table representation. A line of the table represents binary inputs of variable (or variables) and the appropriate binary result of arithmetic calculation with input (or inputs) and the constant. The constant does not change its values independently to a value of input. This fact allows to do not specify a constant value at all, but it is counted by default for every line of the truth table.

Let's consider multiplication of the 5-bit constant 19 on 4-bit variable X by 4-bit modulo 13. The central part of Table 1 is the truth table and describes $(19 \cdot X)\ (mod\ 13) = R$. It consists of 13 lines, because X varies from 0 to 12 according to the modulo 13. As well the modulo value defines 4-bit result of multiplication.

Table 1: Truth table for $(X \cdot 19)\ (mod\ 13) = R$

decimal X	x_4	x_3	x_2	x_1	r_4	r_3	r_2	r_1	$(X \cdot 19)\ (mod\ 13) = R$
0	0	0	0	0	0	0	0	0	$(0 \cdot 19)\ (mod\ 13) = 0$
1	0	0	0	1	0	1	1	0	$(1 \cdot 19)\ (mod 13) = 6$
2	0	0	1	0	1	1	0	0	$(2 \cdot 19)\ (mod 13) = 12$
3	0	0	1	1	0	1	0	1	$(3 \cdot 19)\ (mod 13) = 5$
4	0	1	0	0	1	0	1	1	$(4 \cdot 19)\ (mod 13) = 11$
5	0	1	0	1	0	1	0	0	$(5 \cdot 19)\ (mod 13) = 4$
6	0	1	1	0	1	0	1	0	$(6 \cdot 19)\ (mod 13) = 10$
7	0	1	1	1	0	0	1	1	$(7 \cdot 19)\ (mod 13) = 3$
8	1	0	0	0	1	0	0	1	$(8 \cdot 19)\ (mod 13) = 9$
9	1	0	0	1	0	0	0	1	$(9 \cdot 19)\ (mod 13) = 2$
10	1	0	1	0	1	0	0	0	$(10 \cdot 19)\ (mod 13) = 8$
11	1	0	1	1	0	0	0	1	$(11 \cdot 19)\ (mod 13) = 1$
12	1	1	0	0	0	1	1	1	$(12 \cdot 19)\ (mod 13) = 7$

2.2 Partitioning Boolean Functions

In this contribution we explore minimized BDDs in the context of arithmetic operations representation. BDD is well know and basic technique of Boolean

functions minimization [28–32]. Generally, it is based on Shannon expression [33]:

$$f(x_1, x_2, \ldots, x_m) =$$
$$\overline{x_i} \cdot f(x_1, x_2, ..., x_{i-1}, 0, x_{i+1}, ..., x_m) \vee x_i \cdot f(x_1, x_2, ..., x_{i-1}, 1, x_{i+1}, ..., x_m).$$

We consider minimized BDD as the reduced ordered BDD (ROBDD) with minimal number of nodes [22, 34]. The resulting ROBDD is split minimized DBB with j-splitting procedure *Parsin* into j-input subfunctions [35], where j is taken according to the demands of custom libraries or number of LUT inputs on FPGA.

Example. Consider $j = 4$ and following functions:

$$f^1 = x_1 \cdot x_2 \cdot \overline{x_4} \cdot x_5 \cdot \overline{x_6} \vee \overline{x_1} \cdot x_4 \cdot \overline{x_5} \cdot x_6 \vee x_2 \cdot \overline{x_3} \cdot x_5;$$
$$f^2 = \overline{x_1} \cdot x_4 \cdot \overline{x_5} \cdot \overline{x_6} \vee \overline{x_1} \cdot \overline{x_3} \cdot x_5 \vee x_1 \cdot x_2 \cdot x_3 \cdot x_5 \cdot \overline{x_6} \vee x_1 \cdot \overline{x_2} \cdot x_4 \cdot \overline{x_5} \cdot x_6;$$
$$f^3 = x_1 \cdot \overline{x_2} \cdot \overline{x_3} \cdot x_6 \vee x_1 \cdot \overline{x_2} \cdot x_4 \cdot x_6 \vee x_1 \cdot \overline{x_3} \cdot x_4 \cdot x_6 \vee$$
$$\overline{x_1} \cdot x_2 \cdot \overline{x_4} \cdot x_5 \cdot \overline{x_6} \vee x_1 \cdot \overline{x_2} \cdot x_5 \vee x_2 \cdot \overline{x_3} \cdot x_5.$$

These functions are represented with the truth table shown in Table 2.

Table 2: Truth table for f_1, f_2, f_3

x_1	x_2	x_3	x_4	x_5	x_6	f_1	f_2	f_3
1	1	-	0	1	0	1	0	0
0	-	-	1	0	1	1	0	0
0	-	-	0	1	0	0	1	0
0	-	0	-	1	-	0	1	0
1	1	1	-	1	0	0	1	0
1	0	-	1	1	1	0	1	0
1	0	0	-	-	1	0	0	1
1	0	-	1	-	1	0	0	1
1	-	0	1	-	1	0	0	1
0	1	-	0	1	0	0	0	1
1	0	-	-	1	-	0	0	1
-	1	0	-	1	-	1	0	1

In the result of BDD-minimization with *BDD-builder* [35] f^1, f^2, f^3 take the following form:

$$\begin{gathered}
f^1 = \overline{x_1} \cdot \psi^1 \vee x_1 \cdot \psi^2, \;\; f^2 = \overline{x_1} \cdot \phi^3 \vee x_1 \cdot \psi^4, \;\; f^3 = \overline{x_1} \cdot \psi_2 \vee x_1 \cdot \psi^6, \\
\psi^1 = \overline{x_2} \cdot \phi^1 \vee x_2 \cdot \phi^2, \qquad \psi^2 = x_2 \cdot \phi^3, \qquad \psi^4 = \overline{x_2} \cdot s^1 \vee x_2 \cdot \phi^4, \\
\psi^6 = \overline{x_2} \cdot \phi^5 \vee x_2 \cdot \phi^6, \;\; \phi^2 = \overline{x_3} \cdot s^2 \vee x_3 \cdot s^1, \;\; \phi^3 = \overline{x_3} \cdot \lambda^3 \vee x_3 \cdot s^4, \\
\phi^4 = x_3 \cdot \lambda^4, \qquad \phi^5 = \overline{x_3} \cdot \lambda^2 \vee x_3 \cdot s^2, \qquad \phi^6 = \overline{x_3} \cdot s^2, \\
s^1 = x_4 \cdot \lambda_1, \qquad s^2 = \overline{x_4} \cdot \lambda^3 \vee x_4 \cdot \lambda^2, \qquad s^4 = \overline{x_4} \cdot \lambda^4, \\
\lambda_1 = \overline{x_5} \cdot \omega^1, \qquad \lambda^2 = \overline{x_5} \cdot \omega^1 \vee x_5, \qquad \lambda^3 = x_5, \\
\lambda^4 = x_5 \cdot \omega^2, \qquad \omega^1 = x_6, \qquad \omega^2 = \overline{x_6}.
\end{gathered} \tag{1}$$

Multilevel representation of (1) is represented on Figure 1.

Intermediate functions $\phi^2, \phi^4, \phi^5, \phi^6, s^4, \lambda^1, \lambda^3, \omega^1, \omega^2$ have been annihilated in the result of j-splitting procedure *Parsin* [35], where $j = 4$. Index j has been taken according to 4-input LUTs. The result of *Parsin* procedure implementation of the representation 1 is transformed into 11 equations shown in the representation 2, which are synthesised into 11 4-input LUTs on FPGA shown on Figure 2.

$$\begin{gathered}
f^1 = \overline{x_1} \cdot \psi^1 \vee x_1 \cdot x_2 \cdot \phi^3, \;\; f^2 = \overline{x_1} \cdot \phi^3 \vee x_1 \cdot \psi^4, \\
f^3 = \overline{x_1} \cdot \psi_6 \vee \overline{x_1} \cdot x_2 \cdot \phi^3, \\
\psi^1 = \overline{s^1} \cdot x_3 \vee x_2 \cdot \overline{x_3} \cdot \overline{s^2} \vee \overline{x_2} \cdot \overline{s_1}, \qquad \phi^3 = \overline{x_3} \cdot x_5 \vee x_3 \cdot (\overline{x_4} \cdot (x_5 \cdot \overline{x_6})), \\
\psi^4 = \overline{x_2} \cdot s^1 \vee x_2 \cdot x_3 \cdot \lambda^4, \;\; \psi^6 = x_2 \cdot \overline{x_3} \cdot \overline{s^2} \vee \overline{x_2} \cdot \overline{x_3} \cdot \overline{s^2} \cdot \lambda^2 \vee x_2 \cdot x_3 \cdot s^2 \\
s^1 = x_4 \cdot (x_5 \cdot \overline{x_6})), \;\; \lambda^4 = x_5 \cdot \overline{x_6} \;\; s^2 = \overline{x_4} \cdot x_5 \vee x_4 \cdot (x_5 \cdot \overline{x_6} \vee x_5); \\
\lambda^2 = x_5 \cdot \overline{x_6} \vee x_5.
\end{gathered} \tag{2}$$

3 Experimental Results

Our interests to constant multiplication arises from multioperands additions:

$$((X_1 \cdot C_1 (mod\ P) + X_2 \cdot C_2 (mod\ P) + \ldots + X_h \cdot C_h (mod\ P))\ (mod\ P)$$

and

$$X_1 \cdot C_1 + X_2 \cdot C_2 + \ldots + X_h \cdot C_h - r \cdot P,$$

where $C_1, C_2, \ldots, C_h$, P, r - natural constants, $X_1, X_2, \ldots X_h$ - natural variables. Both representations or parts of them are basic in neural networks, image recognition, cryptography, etc. In the context of RNS $C_1, C_2, \ldots C_h$ and P achieve thousand bits, at the same time $X_1, X_2, \ldots X_h$ may vary from 5 bits to the bit-range of $C_1, C_2, \ldots C_h$.

We demonstrate the results of the synthesis of constant multipliers on FPGA and ASIC in standard arithmetic and by modulo for four bit-ranges of X, four values of constants C and four moduli P, where

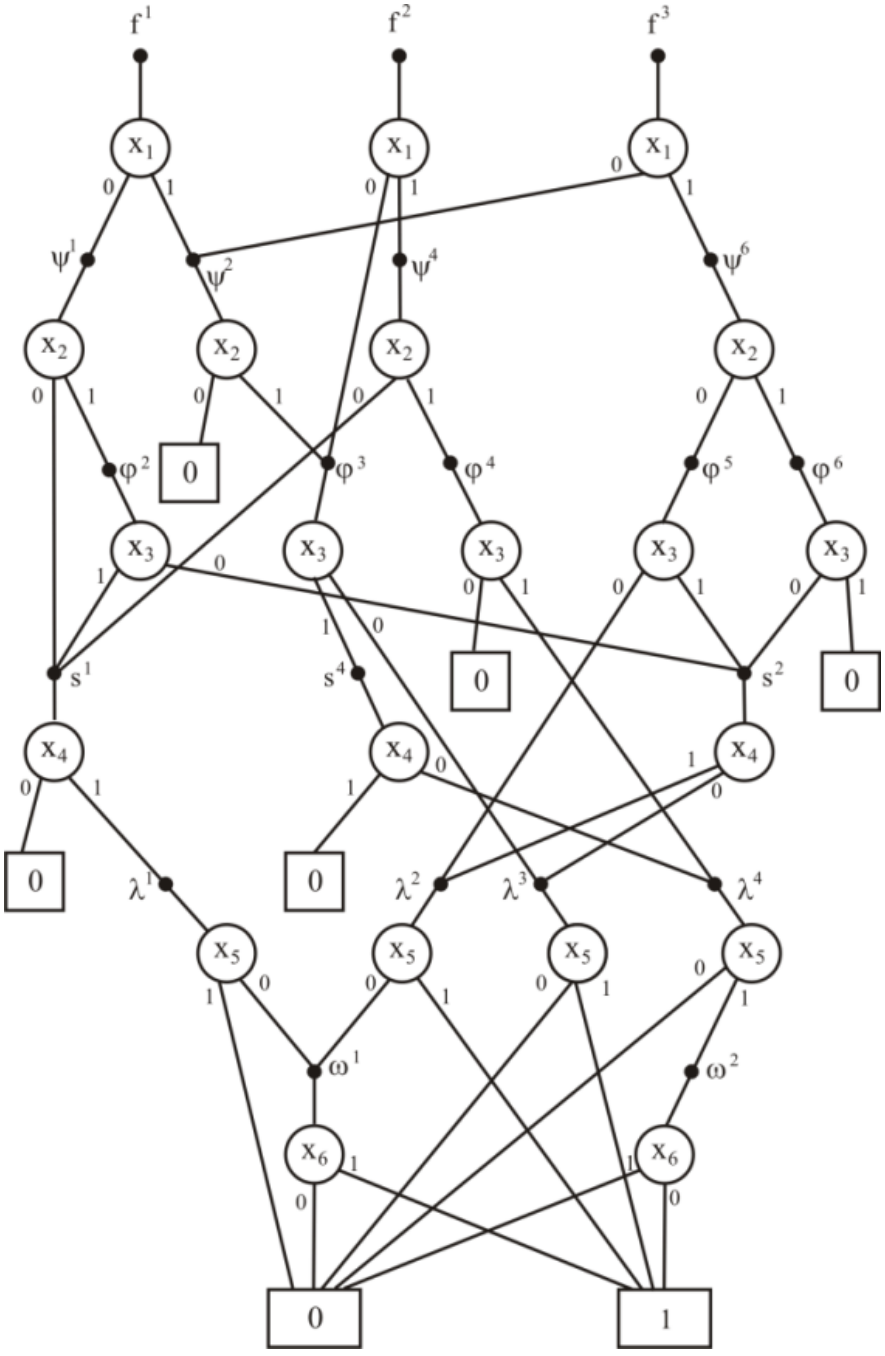


Fig. 1: Multilevel representation of (1) by BDD

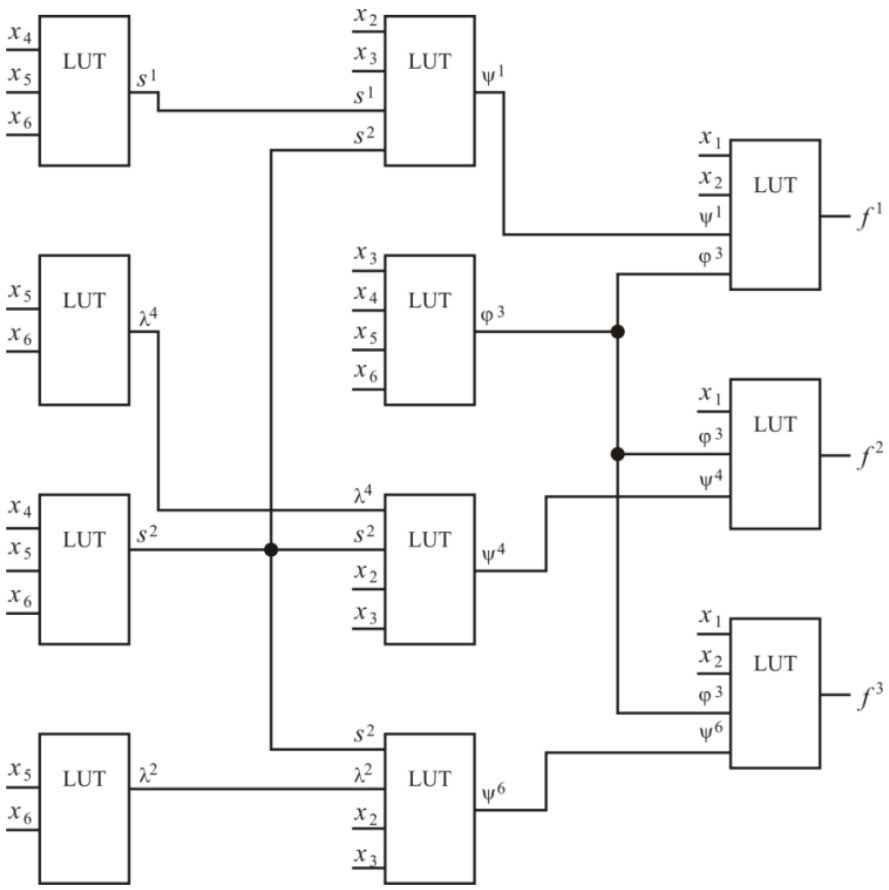


Fig. 2: LUTs realization of the representation (2)

1. 5-bit variable X, 33-bit constant $C = 4653525600$, and 33-bit modulo $P = 4974458400$. We titled it as $\mathbf{5 \times 33}$ for standard multiplication $X \times C$ and $\mathbf{5 \times 33}$ (**33**) for multiplication by modulo $X \times C$ $(mod\ P)$ on the resulting plots;
2. 7-bit variable X, 64-bit constant $C = 15924368328194956800$, and 65-bit modulo $P = 19437096635885020800$. We titled it as $\mathbf{7 \times 64}$ for standard multiplication $X \times C$ and $\mathbf{7 \times 64}$ (**65**) for multiplication by modulo $X \times C$ $(mod\ P)$ on the resulting plots;
3. 8-bit variable X, 128-bit constant $C = 324167785992983736249587182700260749056$, and 129-bit modulo $P = 344055380225682124976555721516227666176$. We titled it as $\mathbf{8 \times 128}$ for standard multiplication $X \times C$ and $\mathbf{8 \times 128}$ (**128**) for multiplication by modulo $X \times C$ $(mod\ P)$ on the resulting plots;
4. 8-bit variable X, 258-bit constant $C = 428092581552246531046308046973043184312835253714261251748800951846523264768000$, and 258-bit modulo

P = 429937808196868283335300754072064922176252647049236515764959576638965175392000. We titled it as $\mathbf{8 \times 258}$ for standard multiplication $X \times C$ and $\mathbf{8 \times 258}$ ($\mathbf{258}$) for multiplication by modulo $X \times C$ ($mod\ P$) on the resulting plots.

We conducted experiments for four sorts of constant multiplication design:

- ***no_per***. Multiplication is represented by truth table with ROBDD minimization in *BDD-builder* [35] without *Parsin*;
- ***per_6***. Multiplication is represented by truth table with ROBDD minimization in *BDD-builder* [35] and further 6-splitting procedure in *Parsin* [35], as described in Section 2.2. The value 6 in the splitting procedure is relevant to 6-input LUTs in the FPGA family we experimented on;
- ***per_12***. Multiplication is represented by truth table with ROBDD minimization in *BDD-builder* [35] and further 12-splitting procedure in *Parsin* [35], as described in Section 2.2;
- ***Vivado***. It is the direct multiplication $X \cdot C = R$ as expression in Verilog.

The experiments are conducted in Xilinx ISE Vivado 15.1 on FPGA and in Leonardo Spectrum by Mentor Graphics in ASIC in custom library *power* (see Table 3).

We measured the performance and the area costs. The performance of multipliers is considered as the delay in nanoseconds (ns) from input to output through the critical path. The area costs of constant multipliers is considered as number of LUTs occupied on FPGA and cells on ASIC according to the custom library *power*.

3.1 Constant Multiplication on FPGA

We experimented in Xilinx Vivado 2015.1 on Artix-7 xc7a200tffv1156-1 with 6-input LUTs and 500 input-output blocks. We switched off implementation of Block RAMs and DSPs by setting up *-max_bram* and *-max_dsp* to 0 in the properties of the synthesis.

Constant Multiplication in Standard Arithmetic on FPGA. The plots on Figures 3 and 4 compare of critical paths and number of occupied LUTs for constant multipliers designed according to the four approaches.

In average *per_6* approach leads to the fastest circuits. The slowest realization is synthesised by the direct multiplication and it is slower approximately on 30%, 15%, 50%, and 20% for 5×33, 7×64, 8×128, and 8×258 respectively.

Fig. 3: Delay of constant multipliers on FPGA Xilinx Artix-7

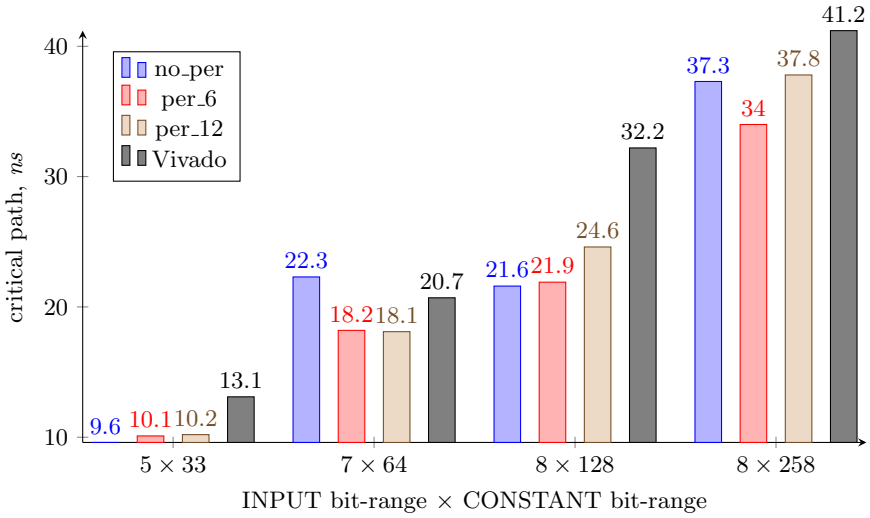
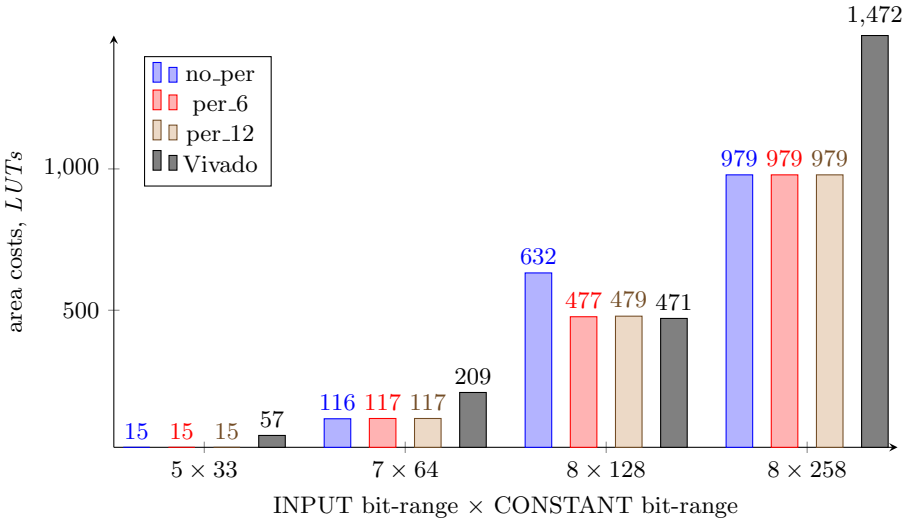


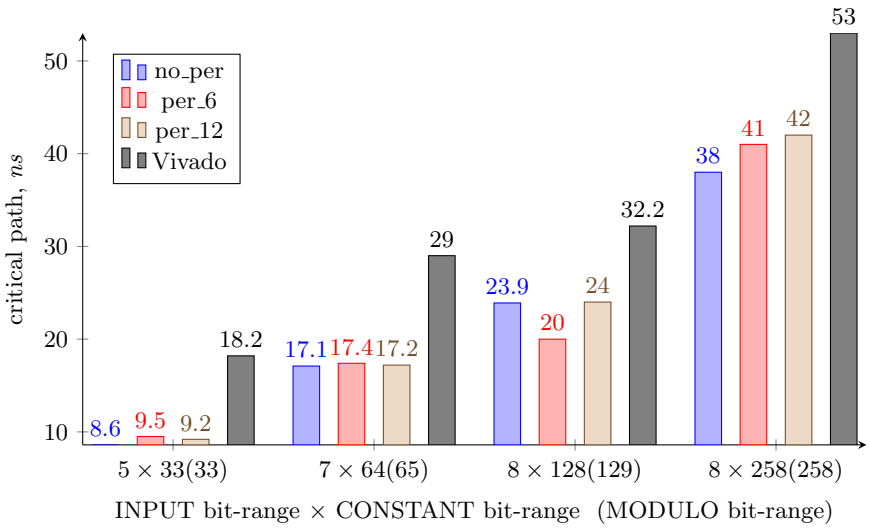
Fig. 4: Area costs of constant multipliers on FPGA Xilinx Artix-7



The area costs of multipliers synthesised by *per_6* and *per_12* are equivalent and occupy the same number of LUTs as *no_per* for 5×33, 7×64, 8×258 and as *Vivado* for 8×128. In the rest cases *per_6* and *per_12* uses less number of LUTs than the direct multipliers in three times for 5×33, in 1.8 times for 7×64, and 1.5 times for 8×258.

Constant Multiplication by Modulo on FPGA. The plots on Figures 5 and 6 compare of critical paths and number of occupied LUTs for constant multipliers by modulo designed by four approaches.

Fig. 5: Delay of constant multipliers by modulo on FPGA Xilinx Artix-7



In average three ROBDD based approaches lead to equivalent performance of multipliers by modulo in all cases and faster than direct multiplication by Vivado approximately in 2 times for $5 \times 33(33)$, 1.7 times for $7 \times 64(65)$, 1.3 times for $8 \times 128(129)$ and $8 \times 258(258)$.

The area costs of three ROBDD based approaches are significantly lower comparing with direct multiplication for $5 \times 33(33)$ in 14 times and $7 \times 64(65)$ in 2.7 times. The direct multiplication $8 \times 128(129)$ occupies on 40% more LUTs than *per_12*, on 10% than *no_per* and on 5% than *per_6*. All approaches demonstrate the equivalent number of LUTs for $8 \times 258(258)$ multiplication.

3.2 Constant Multiplication in Standard Arithmetic on ASIC

Leonardo Spectrum provides custom and external libraries implimentation option. We conducted experiments on the custom library *power* provided by Hi-Tech factory *"Integral"* (Minsk, Belarus). The list of combinational functions of the library is shown in Table 3.

Fig. 6: Area costs of constant multipliers by modulo on FPGA Xilinx Artix-7

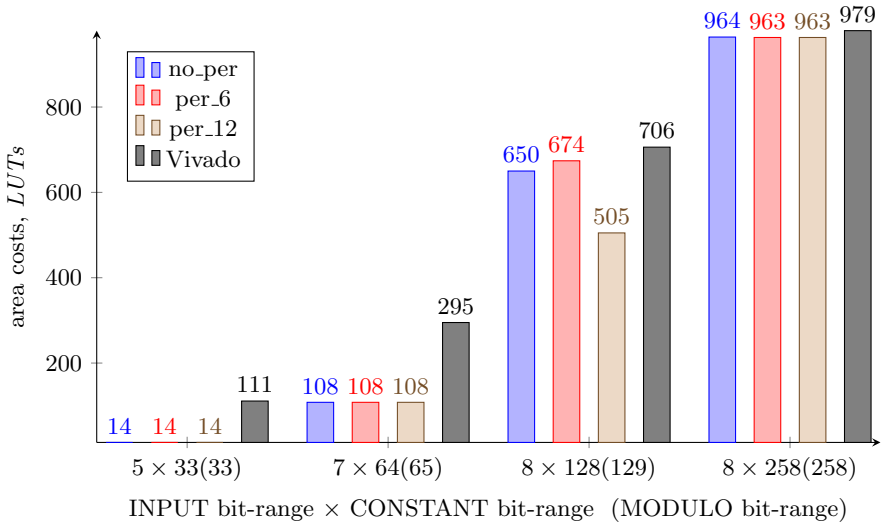


Table 3: Logic elements of the custom library *power*

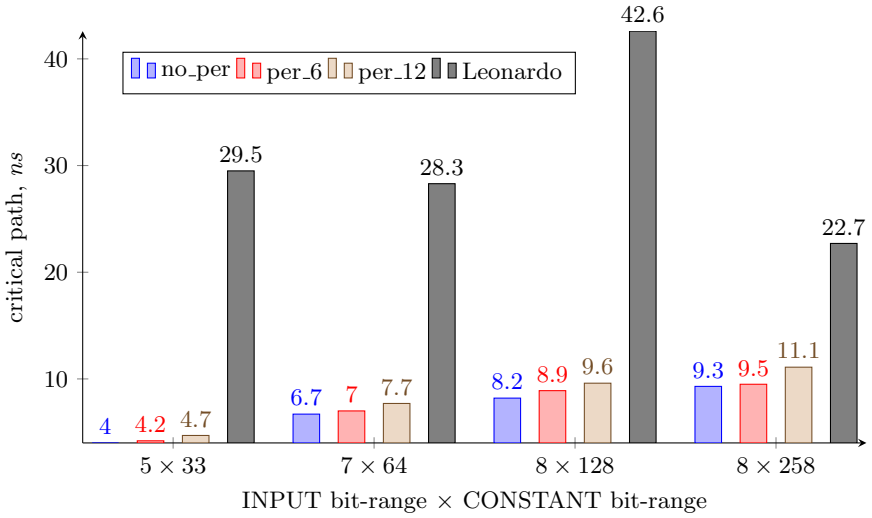
Title of logic element	Function of logic element
N	$f = \overline{x_1}$
NA	$f = \overline{x_1 \cdot x_2}$
NA3	$f = \overline{x_1 \cdot x_2 \cdot x_3}$
NA4	$f = \overline{x_1 \cdot x_2 \cdot x_3 \cdot x_4}$
NO	$f = \overline{x_1 \vee x_2}$
NO3	$f = \overline{x_1 \vee x_2 \vee x_3}$
NO4	$f = \overline{x_1 \vee x_2 \vee x_3 \vee x_4}$
NOA	$f = \overline{x_1 \cdot x_2 \vee x_3}$
NAO	$f = \overline{(x_1 \vee x_2) \cdot x_3}$
NOA3	$f = \overline{x_1 \cdot x_2 \cdot x_3 \vee x_4}$
NAO3	$f = \overline{(x_1 \vee x_2 \vee x_3) \cdot x_4}$
NO3A	$f = \overline{x_1 \cdot x_2 \vee x_3 \vee x_4}$
NA3O	$f = \overline{(x_1 \vee x_2) \cdot x_3 \cdot x_4}$
NOAA	$f = \overline{x_1 \cdot x_2 \vee x_3 \cdot x_4}$
NAOO	$f = \overline{(x_1 \vee x_2) \cdot (x_3 \vee x_4)}$
NO3AA	$f = \overline{x_1 \cdot x_2 \vee x_3 \cdot x_4 \vee x_5}$
NA3OO	$f = \overline{(x_1 \vee x_2) \cdot (x_3 \vee x_4) \cdot x_5}$
NA3O3	$f = \overline{(x_1 \vee x_2 \vee x_3) \cdot x_4 \cdot x_5}$
NO3A3	$f = \overline{x_1 \cdot x_2 \cdot x_3 \vee x_4 \vee x_5}$
NO3AAA	$f = \overline{x_1 \cdot x_2 \vee x_3 \cdot x_4 \vee x_5 \cdot x_6}$
NA3OOO	$f = \overline{(x_1 \vee x_2) \cdot (x_3 \vee x_4) \cdot (x_5 \vee x_6)}$

In order to achieve the maximum performance (i.e. the shortest critical path) we set up *delay optimization* and *optimize longest path (no constrains)* in the list of options for the synthesis.

The plots on Figures 7 and 6 compare of critical paths and elementary cells for constant multipliers designed by four approaches.

Any arithmetic operation by modulo are not synthesizable in Leonardo Spectrum, thus we compared of four approaches only for direct multiplication.

Fig. 7: Delay of constant multipliers in custom library *power*



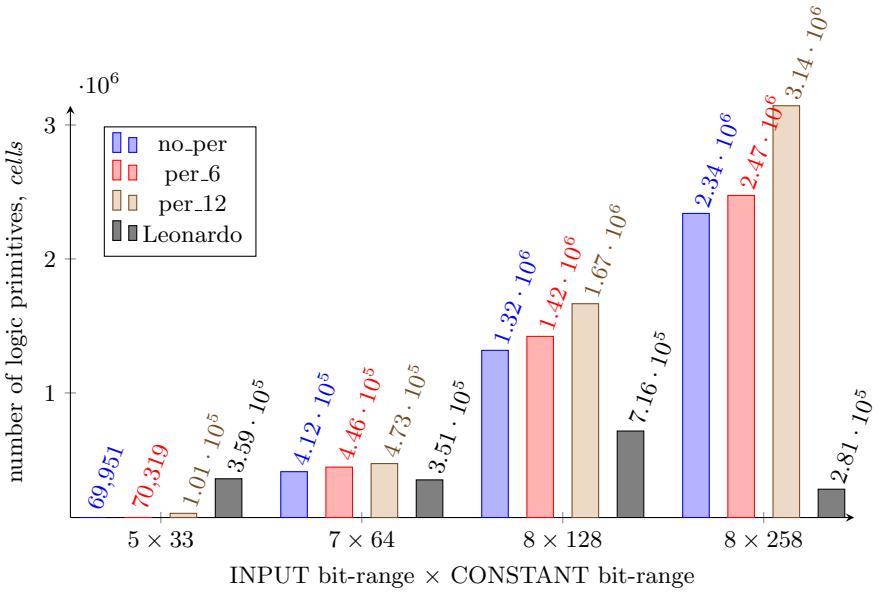
In average all three approaches based on ROBDD provide significantly faster data processing then the direct CM: in 7 times for 5×33, in 4 times 50%, in 2 times for 7×64, in 5 times for 8×128, in 2 times for 8×258.

The multipliers based on ROBDD minimization without *Parsin* (*no_per*) and with 6-splitting procedure in *Parsin* have more or less equivalent area costs for all bit-ranges with minor advantage by (*no_per*). Multipliers based on 12-splitting procedure occupy to up to 1.5 times more cells than both others. Unexpected and in opposite to the performance the area costs of direct multipliers exceed in 3-5 times other three types only for 5×33. In the rest cases direct multipliers smaller in 1.3 times for 7×64, in 2 times for 8×128, and significantly smaller in 9 times for 8×258. Note curious fact that the area costs of direct multiplier for 8×258 is smaller than even direct multipliers 7×64 and 8×128.

4 Conclusions and Further Research

We considered three bit ranges of CM for small variable inputs (5-, 7-, 8-bit) and quite big constants (33-, 64-, 128-, and 258-bit) for standard multiplication,

Fig. 8: Area costs of constant multipliers in custom library *power*



as well as multiplication for 33-, 65-, 129-, 258-bit moduli. We propose to use Boolean representation and minimization to design efficient multipliers in performance and area costs. The synthesis on FPGA shown up to 2 times advantage in performance and up to 5 times advantage in area costs under direct multiplication. Performance of multipliers based on Boolean minimization in ASIC achieves 7 times and area costs is smaller in one case only.

Our further research will involve implementation of Boolean minimization for multioperand addition unit

$$X_1 \cdot C_1 + X_2 \cdot C_2 + \ldots + X_h \cdot C_h - r \cdot P$$

and multifunctional floating point computation unit

$$A \cdot B + C$$

for half-, single- and double-precision calculation.

References

1. M.J. Wirthlin and B. McMurtrey, “Efficient Coefficient Multiplication Using Advanced FPGA Architectures”, in Field-Programmable Logic and Applications. Proceedings of the 11th International Workshop, FPL 2001, Lecture Notes in Computer Science, Springer-Verlag, Aug. 2001, pp. 555-564.

2. P. Tummeltshammer, J.C. Hoe, and M. Püschel, "Time-Multiplexed Multiple-Constant Multiplication", IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 26, No. 9, 2007, pp. 1561-1563.
3. A.G. Dempster and M.D. Macleod, "Generation of Signed-Digit Representations for Integer Multiplication", IEEE Signal Procesing letters, Vol. 11, No. 8, 2004, pp.663-665.
4. Y. Voronenko and M.S. Püschel, "Multiplierless Multiple Constant Multiplication", ACM Transactions on Algorithms, Vol. 3, No. 2, 2007, pp. 1-38.
5. A. Volkova, M. Istoan, F. de Dinechin, and T. Hilaire, "Towards Hardware IIR Filters Computing Just Right: Direct Form I Case Study", IEEE Transactions on Computers, Vol. 68 , Issue 4, 2019, pp. 597-608.
6. O. Gustafsson, A.G. Dempster, K. Johansson, M.D. Macleod, and L. Wanhammar, "Simplified Design of Constant Multipliers", Circuits System Signal Processing, Vol. 25, No. 2, 2006, PP. 225–251.
7. V. Dimitrov, L. Imbert, and A. Zakaluzny, "Multiplication by a Constant is Sublinear", 18th IEEE Symposium on Computer Arithmetic (ARITH '07), Montepellier, France, 25-27 June 2007, pp. 261-268.
8. 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.
9. J. Faraone, M. Kumm, M. Hardieck, L. Xueyuan, D. Boland, and P.H.W. Leong,"Deep Neural Networks using FPGA-Optimized Multipliers", IEEE Transactions on Very Large Scale Integration Systems (TVLSI), 28(1), 2020, pp. 115-128.
10. M. Kumm, "Optimal Constant Multiplication using Integer Linear Programming", IEEE Transactions on Circuits and Systems II: Express Briefs, Vol. 65, Issue 5, May 2018.
11. 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.
12. P.V.A. Mohan, "Residue Number System. Theory and applications", Springer International Publishing, 2016, 351 p.
13. P. Martins and L. Sousa, "The Role of Non-Positional Arithmetic on Efficient Emerging Cryptographic Algorithms", IEEE Access, 2020, pp. 59533-59549.
14. E.E. Swartzlander, "Merged Arithmetic", IEEE Transactions on Computers, Vol. 29, 1980, pp. 946–950.
15. M. Ercegovac, "On Approximate Arithmetic", Proceedings of the 47-th Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, CA, USA, 2013, pp. 126–130.
16. A.R. Omondi, "Cryptography Arithmetic. Algorithms and Hardware Architectures", Springer Nature Switzerland, 2020.
17. "Computers, Software, Engineering and Digital Devices", Ed. R.C. Dorf. – Taylor and Francis, 2006, 576 p.
18. A. Karatsuba and Y. Ofman, "Multiplication of Many-Digital Numbers by Automatic Computers", In: Proceedings of the USSR Academy of Science. Vol. 145. 2. In Russian. USSR, 1962, pp. 293–294.
19. A. Schönhage and V. Strassen, "Fast Multiplication of Large Numbers", Computing 7.3 (1971). In German: Schnelle Multiplikation großer Zahlen, pp. 281–292.
20. A.D. Booth, "A Signed Binary Multiplication Technique", The Quarterly Journal of Mechanics and Applied Mathematics. IV (2), pp. 236–240.

21. P.L. Montgomery, "Modular Multiplication without Trial Division Mathematics of Computation", Mathematics of Computation, Vol. 44, No. 170. Apr., 1985, p. 519-521.
22. D.A. Knuth, "Art of Computer Programming, Volume 4A, The: Combinatorial Algorithms, Part 1", Addison-Wesley Professional, 2014.
23. https://embedded.eecs.berkeley.edu/pubs/downloads/espresso
24. L. Amaru, "New Data Structures and Algorithms for Logic Synthesis and Verification", Springer, 2016.
25. P. Bibilo, L. Cheremisinova, S. Kardash, N. Kirienko, V. Romanov, and D. Cheremisinov, "Automatizations of the logic synthesis of CMOS circuits with low power consumption", Programnaia ingeniria, 2013, Vol.8, pp. 35-41 (in Russian).
26. D. Gorodecky, "Design of Multipliers Using Fourier Transformations", Further Improvements in the Boolean Domain, Cambridge Scholars Publishing, UK, 2018, Section 3.4, pp. 240-252.
27. D. Gorodecky and T. Villa, "Efficient Hardware Operations for the Residue Number System by Boolean Minimization", Advanced Boolean Techniques, Springer Nature Switzerland, 2019, Section 11, pp. 237-258.
28. C. Y. Lee, "Representation of Switching Circuits by Binary-Decision Programs", Bell Systems Technical Journal, 38:985-999, 1959.
29. R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams", ACM Computing Surveys, Vol. 24, No. 3 (September, 1992), pp. 293—318.
30. B. Becker and Rolf Drechsler, "Binary Decision Diagrams: Theory and Implementation", Springer, 1998.
31. T. Sasao, "Switching Theory for Logic Synthesis", Springer, 1999.
32. T. Sasao and J.T. Butler, "Applications of Zero-suppressed Decision Diagrams", Morgan & Claypool Publishers, 2014.
33. C.E. Shannon, "A symbolic analysis of relay and switching circuits", Electrical Engineering 57 (12), 1938, pp. 713-723.
34. R.E. Bryant, C. Meinel, "Ordered Binary Decision Diagrams", Logic synthesis and verification. Boston, Dordrecht, London: Kluwer Academic Publishers, 2002. – P. 285–307.
35. P. Bibilo and Yu. Lankevich, "Minimizing the multilevel representations of systems of Boolean functions based on Shannon decomposition", Informatika, N 2, 2017, pp. 45-57.