Efficient Hardware Operations for the Residue Number System by Boolean Minimization

Chapter · January 2020

DOI: 10.1007/978-3-030-20323-8_11

2 authors:

Danila Gorodecky

14 PUBLICATIONS **22** CITATIONS

SEE PROFILE

Tiziano Villa

University of Verona

169 PUBLICATIONS **3,213** CITATIONS

SEE PROFILE

Efficient hardware operations for the residue number system by Boolean minimization

Danila Gorodecky[1] and Tiziano Villa[2]

[1] National Academy of Science of Belarus, Minsk, Belarus
danila.gorodecky@gmail.com
[2] University of Verona, Verona, Italy
tiziano.villa@univr.it

1 Introduction

The idea of the Residue Number System (RNS) goes back to an ancient Chinese source showing how to convert residues into numbers, and was later formalized by C.F. Gauss in the 19th century. Since the advent of digital computers, there have been many papers proposing algorithms to implement efficiently RNS on computers.

The main advantage of RNS is the speed and reliability of arithmetic computations [1–3]. The first application of RNS was in the search of prime numbers. Nowadays implementations of RNS can be found in anti-aircraft systems [4], neural computations [1], real-time signal processing (pattern recognition) [5], cryptography [6]. Modular arithmetic (MA) is effective for processing large data flows (with several hundreds or thousands bits) [7]. So RNS allows to increase significantly hardware performance and to upgrade reliability and noise immunity in signal processing and data transferring. A conference was held in 2005 in Russia on the 50th anniversary of the introduction of RNS in scientific computations [22], and it was reported on the key role of RNS in radars, in space and military aircraft (e.g., Sukhoi) data transferring, and in other important technologies.

This contribution describes an efficient combinational hardware computation of modular multiplication and of the modulus function ($X(mod\ P)$) for an arbitrary modulo. We report also experimental results and compare with industrial tools.

2 Basic knowledge about RNS

The Chinese reminder theorem [23] states that there is a one-to-one correspondence between a set of residues $X_1, X_2, ..., X_n$ and a number from 0 to $p_1 \cdot p_2 \cdot ... \cdot p_n - 1 = P - 1$. Since the value of the represented number is invariant under any permutation of the residues, RNS is a non-positional number system.

These features make RNS an alternative system with pros and cons compared with other systems of representation. For example, from one side it allows to parallelize computations and hence to speed them up, on the other side it does

not allow to compare two numbers represented by their residues and figure out which one is greater without additional operations like backward conversion into the positional system.

RNS is a form of parallel data processing, where computer arithmetic is performed using the residues of the division by a pre-selected base of co-primes moduli $\{p_1, p_2, ..., p_m\}$. The residues have a lower number of digits than the original numbers and arithmetic operations over the residues can be performed separately for each modulo of the base, resulting in faster processing (e.g., faster addition and multiplication), compared to other forms of parallel data processing. The parallelism is achieved by computing on the residues. The residues $(A_1, B_1, A_2, B_2, ..., A_n, B_n)$ are the results of division of the input numbers (A and B) by s pre-selected set of co-primes $(p_1, p_2, ..., p_n)$ - moduli, where $p_1 \cdot p_2 \cdot ... \cdot p_n = P$. Computing in RNS is not restricted to arithmetic operations with two operands, and it is suitable for an arbitrary number of operands.

Data processing in RNS includes the following steps. First, input operands $A_1, A_2, \ldots, A_n$ are converted from positional to modular representations computing the remainders (or residues) with respect to the moduli $\{p_1, p_2, \ldots, p_m\}$ (see left block in Fig. 1); then arithmetic operations over the residues of the operands for each modulo p_i, where $i = 1, \ldots, n$, are computed (middle block in Fig. 1); finally, the results $S_1, S_2, ..., S_m$ for each modulo are converted back from modular to positional representations S (see right block in Fig. 1).

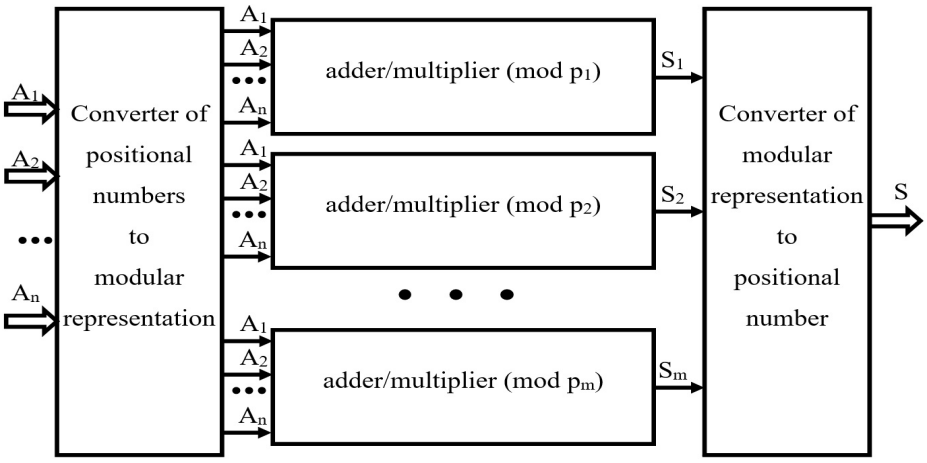


Fig. 1: Common structure of RNS.

Conversion into modular representation (direct conversion) is realized by the modulo $X(mod\ P)$ function, whose result is fed into the second step of operations. The second step of the RNS computation requires performing modular summation, multiplication, and other arithmetic functions such $A \cdot B + C$. The third step in RNS computes the polynomial form $S_1 \cdot C_1 + S_2 \cdot C_2 + ... + S_m \cdot$

$C_m - P \cdot r$, where $S_1, S_2, ...$ are outputs of the previous step, $C_1, C_2, ...$ are precalculated constants, r is a constant which is obtained during the computation of the polynomial, and $P = p_1 \cdot p_2 \cdot ... \cdot p_m$. In other words the third step in RNS computes $(S_1 \cdot C_1 + S_2 \cdot C_2 + ... + S_m \cdot C_m)(mod\ P)$. Therefore, the main arithmetic operations needed for RNS computations are the modulo function $X(mod\ P)$, modular summation, and modular multiplication.

For instance, $A = 37$, $B = 19$, and $P = p_1 \cdot p_2 \cdot p_3 = 3 \cdot 5 \cdot 7 = 105$. In this case $A_1 = 1$, $B_1 = 1$, $A_2 = 2$, $B_2 = 4$, $A_3 = 2$, $B_3 = 5$, i.e. $A = (1, 2, 2)$ and $B = (1, 4, 6)$. In this case, addition is produced by summing residues with the appropriate index, i.e. $A + B = ((A_1 + B_1)\ mod\ p_1, (A_2 + B_2)\ mod\ p_2, (A_3 + B_3)\ mod p_3) = (2, 1, 0) = S$.

There are a couple of ways to convert a number S into a positional number. The most common one is based on the following formula:

$$\begin{aligned} Z =& (X_1 \cdot Y_1 + X_2 \cdot Y_2 + ... + X_n \cdot Y_n)\ mod\ P \\ =& X_1 \cdot Y_1 + X_2 \cdot Y_2 + ... + X_n \cdot Y_n - k \cdot P, \end{aligned} \tag{1}$$

where k is a natural number and $Y_i = \frac{P}{p_i} \cdot q$, where q should satisfy the condition $q \cdot \frac{P}{p_i}(mod\ p_i) = 1$, $i = 1, 2, ..., n$ and $q = 1, 2, ..., p_i - 1$.

By this formula

$$S = 2 \cdot 70 + 1 \cdot 21 + 0 \cdot 15 = 161 - 1 \cdot 105 = 56,$$

where: $Y_1 = \frac{105}{3} \cdot 2$, because $2 \cdot \frac{105}{3}(mod 3) = 1$; $Y_2 = \frac{105}{5} \cdot 1$, because $1 \cdot \frac{105}{5}(mod 5) = 1$; $Y_3 = \frac{105}{7} \cdot 1$, because $1 \cdot \frac{105}{7}(mod\ 7) = 1$.

There are a couple of significant limits of RNS implementation for wide range computing, namely, the conversion from the positional system to RNS and backward. In fact, no one electronic design automation (EDA) tool (but Synopsys) can generate a circuit to compute the modulus function. The exception is when modulus $P = 2^\delta$, where δ is a natural number, because an arbitrary number with modulus 2^δ equals the δ least significant bits of this number. Otherwise, the problem is computationally hard. Backward conversion is slightly easier, because it consists of multipliers and summators (as shown in formula (1)). But, eventually, either the modulus function with moodulo P must be computed or a comparison with the value $k \cdot P$ must be performed many times in order to find k, or the conventional mixed-radix approach [10] must be applied, in all cases decreasing significantly the performance of the conversion. In practice both problems are solved by choosing from a small set of special moduli, which limits the applicability of RNS.

3 Computation of the modulo function

The modulus function calculation ($X(mod\ P)$) is a basic arithmetic operation in cryptography and in the residue number system (RNS). In both cases one must handle huge numbers X with hundreds and thousands of bits. However, there

is a significant difference between the two areas in the requirements to compute $X(mod\, P)$. The value of P in cryptography is a prime number or one which can be factored with 2 or rarely 3 factors, i.e. $P = p_1 \cdot p_2 \cdot p_3$, where p_1, p_2, p_3 are prime numbers. In RNS, P can be factored with n factors, where n can be of the order of a few dozens i.e. $P = p_1 \cdot p_2 \cdot ... \cdot p_n$. Despite this difference, the hardware implementation of the modulus function is a bottleneck for both areas.

A major limitation when processing large numbers in RNS is the complexity of hardware realization of converters (left and right blocks in Fig. 1). This is due to the fact that to compute the modulus function and to recover the positional representation one should perform division, modular multiplication, and comparison. There are different approaches to solve this problem (e.g. [1, 3, 8]), but, mostly, they are restricted with respect to the modular values (e.g., $mod\ 2^k - 1, mod\ 2^k, mod\ 2^k + 1$) and to the number of operands.

Cryptography demands that the factors of P be as hard as possible. This fact significantly increases the complexity of hardware realization, since an efficient hardware implementation of $X(mod\, P)$ for an arbitrary P is unknown. In RNS all factors of P are known. Moreover $p_1, p_2, ..., p_n$ are selected as special numbers for which it is known an efficient hardware realization. But the problem is that the set of special numbers, for which efficient algorithms are known, is very limited. This is the main fact which restricts wide RNS deployement.

There are exist [9] approaches for $X(mod\, P)$ design in hardware, but they are sequential or/and they exhibit high hardware costs and low performance compared with approaches for special number sets.

A unit for modulus function computation can be designed with sequential elements, but they require bigger areas and they are slower compared with combinational approaches. On the other hand, the availability of memory allows to compute $X(mod\, P)$ for arbitrary value of P and to pipeline the computation. Sequential realizations may store pre-calculated values of the modulus function [9, 14], or be computed by an automaton model [16], or they may resort to pipelining using a chain of homogeneous arithmetic blocks [11]. A pipelining model is based on combinational logic, where pipelining stages are separated by triggers (latches). More recent designs avoid the use of memory elements and use combinational logic to a large extent, and shortly we will consider some of them.

3.1 Approach based on reducing the input bit-by-bit

In [11], the authors proposed a sequence made by a chain of homogeneous arithmetic blocks, and combinational architectures for pseudorandom number generator. This yields a modulus function architecture based on the following representation:

$$\begin{aligned} X &= P \cdot Q + R = \\ &= P \cdot 2^{\delta} \cdot q_{\delta} + P \cdot 2^{\delta-1} \cdot q_{\delta-1} + \ldots + P \cdot 2^{0} \cdot q_0 + R \end{aligned} \tag{2}$$

and $X(mod\ P) = R$, where $X = (x_\psi, x_{\psi-1}, \ldots, x_1)$ and δ is defined by the inequality $P \cdot 2^{\delta+1} < 2^{\psi} - 1 \leq P \cdot 2^{\delta}$. Notice that P can be an arbitrary number.

Every computational block executes comparison, multiplexing, multiplication by a constant, and subtraction.

For instance, $X = (x_{10}, x_9, ..., x_1)$ and $P = 21$, hence $\delta = 5$. In this case (2) takes the following form:

$$\begin{aligned} X = 21 \cdot Q + R = \\ &= 21 \cdot 2^5 \cdot q_5 + 21 \cdot 2^4 \cdot q_4 + 21 \cdot 2^3 \cdot q_3 + \\ &+ 21 \cdot 2^2 \cdot q_2 + 21 \cdot 2^1 \cdot q_1 + 21 \cdot 2^0 \cdot q_0 + R. \end{aligned}$$

This representation consists of seven addends, where the last R is the result of a modulus function computation and the remaining six addends represent an arithmetic unit. We assign $X = X_5$ as input of the first unit; X_4 is the output of the first unit and the input of the second unit; X_3 is the output of the second unit and the input of the third unit; X_2 is the output of the third unit and the input of the fourth unit; X_1 is the output of the fourth unit and the input of the fifth unit; X_0 is the output of the fifth unit and input of the sixth unit; R is the output of the sixth unit and the result of the $X(mod\ 21)$ computation.

Assume that $X = 888$. Thus $X = 888\ (mod\ 21)$ will be computed by the following six steps:

- as $X_5 \geq 21 \cdot 2^5$, i.e. $888 \geq 672$, then $X_4 = 888 - 21 \cdot 2^5 = 216$;
- as $X_4 < 21 \cdot 2^4$, i.e. $216 < 336$, then $X_3 = 216$;
- as $X_3 \geq 21 \cdot 2^3$, i.e. $216 \geq 168$, then $X_2 = 216 - 21 \cdot 2^3 = 48$;
- as $X_2 < 21 \cdot 2^2$, i.e. $48 < 84$, then $X_1 = 48$;
- as $X_1 \geq 21 \cdot 2^1$, i.e. $48 \geq 42$, then $X_0 = 48 - 21 \cdot 2^1 = 6$;
- as $X_0 < 21 \cdot 2^0$, i.e. $6 < 21$, then $R = 6$.

This approach can be pipelined (1-dimensional) very efficiently by including triggers between homogeneous blocks; morevover, it can be simplified by optimizing the arithmetic operations in every block [15] and organizing a 2-dimensional systolic matrix.

3.2 Approach based on the periodic property of powers of two

Another approach is based on using the periodic property of residues of $2^k(mod\ P)$ [9, 17, 18]. Denoting $2^j \equiv 1(mod\ P)$, it is known that $2^{\alpha \cdot j + i} \equiv 2^i(mod\ P)$, if value α is the period of the modulus P. It means that an n-bit input can be split into j α-bit vectors starting from the least significant bits. The value "j" is "order" and can be $P - 1$ or less. Considering the example from [9], let $P = 19$ and $X = 898871661711_{10} = 000101001110110110110001010011101101101 1_2$, then $\alpha = 18$. Thus, the three (as $\alpha = 18$) 18-bit vectors (adding 14 bits as the most significant to the third vector) can be added to obtain:

$$\begin{array}{l} 00\ 0000\ 0000\ 0000\ 0001 \\ 01\ 0011\ 1011\ 0110\ 1100 \\ \underline{01\ 0100\ 1110\ 1101\ 1011} \\ 10\ 1000\ 1010\ 0100\ 1000 \end{array}$$

This corresponds to 166472. The residue of this 18-bit number can be obtained next by adding the residues of various powers of $2(mod\ 19)$. In short, the periodic property of

$2^k mod\ m$ has been used to simplify the computation. Further simplification is possible for moduli satisfying the property $2^{(P-1)/2}\ (mod\ P)$. Considering $P = 19$, we observe that $2^9 = -1(mod\ 19)$, $2^{10} = -2(mod\ 19)$, $\ldots$, $2^{17} = 10(mod\ 19)$, $2^{18} = 1(mod\ 19)$ and $2^{19} = 2(mod\ 19)$, etc. [9]. Thus,the residues in the upper half of a period are opposite in sign to those in the lower half of the period. Denoting the successive words of half period length as $W_0, W_1, ..., W_\alpha$, where α is odd, we need to estimate $\big(\sum_{i=0}^{(\alpha-1)/2} W_{2i} - \sum_{i=0}^{(\alpha-1)/2} W_{2i+1}\big)$ [9]. For the same example we first divide the given word into 9-bit fields starting from the least significant bits as follows:

$$W_4 = 0001$$
$$W_3 = 0\ 1001\ 1101$$
$$W_2 = 1\ 0110\ 1100$$
$$W_1 = 0\ 1010\ 0111$$
$$W_0 = 0\ 1101\ 1011$$

Then, adding W_0, W_2, W_4 we get $S_e = 10\ 0100\ 1000$ and adding W_1, W_3 we get $W_0 = 1\ 0100\ 0100$. Subtracting S_o from S_1 we have $S = 0001\ 0000\ 0100$. The word lengths of S_o and S_e can be more than $j/2$ bits depending on the number of $j/2$-bit fields in the given binary number. The residue of the resulting word can be found easily with another stage by applying the periodic property and a final $(mod\ P)$ reduction described earlier.

3.3 Approach based on modular exponentiation

In this approach the various residues of powers of 2 (i.e. $2^x(mod\ p_i)$) are obtained using logic functions [19, 20]. Consider as an example $2^{s_3s_2s_1s_0}(mod\ 13)$ from [9]. This expression can be rewritten as:

$$2^{s_3s_2s_1s_0}(mod\ 13) = 2^{8s_3+4s_2}4^{s_1}2^{s_0}(mod\ 13) = 256^{s_3}16^{s_2}4^{s_1}2^{s_0}(mod\ 13) =$$
$$(255s_3 + 1)(15s_2 + 1)4^{s_1}2^{s_0}(mod\ 13) = (3s_3s_2 + 8s_3 + 2s_2 + 1)4^{s_1}2^{s_0}(mod\ 13).$$

Then we can evaluate the bracketed term for various values of $s_0, s_1,$, e.g., for $s_1 = 0, s_0 = 0$, we have $2^{s_3s_2s_1s_0}(mod\ 13) = (3s_3s_2 + 8s_3 + 2s_2 + 1)(mod\ 13)$. Afterwards, given the four assignments $11, 10, 01, 00$ for bits s_3 and s_2, the expression $2^{s_3s_2s_1s_0}(mod\ 13)$ assumes the values $1, 9, 3, 1$, respectively. Finally, the logic function g_0 can be used to represent $2^{s_3s_2s_1s_1}(mod\ 13)$ for $s_1 = 0, s_0 = 0$, according to the assignments of s_3 and s_2, as

$$g_0 = 8s_3\overline{s_2} + 2\overline{s_3}s_2 + 1.$$

In the same manner, the other functions corresponding to s_1, s_0 i.e. 01, 10, 11 can be obtained as

$$g_1 = 4(s_2 \oplus s_3) + 2(\overline{s_3} \oplus s_2) + s_3\overline{s_2},$$
$$g_2 = 8(s_2 \oplus s_3) + 4(\overline{s_3} \oplus s_2) + 2s_3\overline{s_2},$$
$$g_3 = 4(s_2 \oplus \overline{s_3}) + 4s_2\overline{s_3} + 2(s_3 \oplus s_2) + (s_3 \oplus s_2).$$

The logic gates that are used to generate the functions g_0, g_1, g_2, g_3 can be shared among the moduli. For instance, $2^{11}(mod\ 13)$ can be obtained from g_3 (since $s_0 = s_1$), by substituting $s_3 = 1$, $s_2 = 0$ as 7.

3.4 Approach based on varying powers of 2

This technique uses the idea that the residues $(mod\ P)$ of various powers of 2 from 1 to 63 can assume values only between 0 and $(P-1)$ [21]. Thus, the number of "1" bits in the residues corresponding to the 64 bits to be added is less, and it is not the end of it: they can be further reduced by rewriting the residues which have large Hamming weight as the sum of a correction term and of a word with a smaller Hamming weight. This will reduce the number of terms (bits) being added. As an illustration, for modulus 29, the various values of $2^x(mod\ 29)$ from 2^0 to 2^{28} are as follows [9]:

$$1, 2, 4, 8, 16, 3, 6, 12, 24, 19, 9, 18, 7, 14, 28, 27,$$
$$25, 21, 13, 26, 23, 17, 5, 10, 20, 11, 22, 15.$$

Given a 64-bit input word, the values repeat again from 2^{29} till 2^{57}, and once more from 2^{58} till 2^{63}, with many of their bits being zero. Consider the residue 27, i.e. $2^{15}(mod\ 29)$ with Hamming weight 4. It can be written as $(x_{15}2^{15})(mod\ 29) = (27 + 2\overline{x_{15}})$ so that when x_{15} is zero, its value is $(27 + 2)(mod\ 29) = 0$. Since 2 has Hamming weight smaller than 27, the number of bits to be added will be reduced. This property applies to residues $19, 28, 27, 25, 21, 13, 26, 23, 11$ and 15. Thus, for a 64-bit input word, in a conventional design, 64 5-bit words should be added in the general case; since many of their bits are zero, when deleting all these zero bits, we would need to add $27, 28, 29, 31$ and 30 bits in various columns. It can be verified that the numbers of bits to be added in each column (corresponding to 2^i for $i = 4, 3, 2, 1, 0$) are as follows (without Hamming weight optimization and with Hamming weight optimization):

- without optimization: 27, 28, 29, 31, 30;
- with optimization: 17, 21, 25, 32, 18.

3.5 Approach based on special moduli of the type $2^n \pm k$

The approach based on special moduli of the type $2^n \pm k$ is the most common technique to design RNS [9, 10, 8]. The are two main reasons to use these values. The first one is due to their simple forward conversion from the positional numerical system to RNS. The second one is that backword conversion from RNS to the positional system can be designed in hardware with smaller costs than using other values of moduli.

Combinational approaches efficient with respect to performance and area exploit special moduli sets [10], which are variations of $2^s \pm v$, where $v = 1, 3, 5$: $\{2^s - 1, 2^s, 2^s + 1\}$, $\{2^{2\cdot s} - 1, 2^s, 2^{2\cdot s} + 1\}$, $\{2^s - 1, 2^{2\cdot s}, 2^s + 1, 2^{s-1} - 1, 2^{s+1} - 1\}$, etc. These moduli take advantage of the following identities (considering moduli $2^k \pm 1$):

$$2^{k\cdot n}(mod\ (2^k - 1)) = 1(mod\ (2^k - 1))$$

and

$$2^{k\cdot n}(mod\ (2^k + 1)) = (-1)^{n-1}(mod\ (2^k + 1)).$$

For instance, when $P = 2^4 + 1$ and $X = 149$, we have:

$$\begin{aligned} &149(mod\ (2^4 + 1)) = (10010101)_2(mod\ 17) = \\ &= ((0101)_2 + (-1)^{2-1}(1001)_2)(mod\ 17) = \\ &= (5 - 9)(mod\ 17)) = -4(mod\ 17) = 13(mod\ 17). \end{aligned}$$

Given that in the RNS representation the moduli must be co-prime numbers, multiplication of two 1000-bit numbers using the moduli $\{2^s-1, 2^{2\cdot s}, 2^s+1, 2^{s-1}-1, 2^{s+1}-1\}$ requires $s \approx 400$ bits, which decreases the computational efficiency of the transformation. The same multiplication can be realized using a set of smaller moduli, since there are more than 400 12-bit numbers that are co-prime. Note that, in order to represent uniquely numbers in RNS, the result of the calculation must not exceed $P = p_1 \cdot p_2 \cdot ... \cdot p_m$. If $P = (2^s - 1) \cdot (2^{2\cdot s}) \cdot (2^s + 1) \cdot (2^{s-1} - 1) \cdot (2^{s+1} - 1)$, then s requires approximately a 400-bit number.

4 Hardware design of functions by modulo

The approach that we propose is characterized as follows:

1. It is valid for an arbitrary modulo and bit range of the inputs.
2. It can be applied to modular multiplication, modular addition, and to the modulo function.
3. It is based on combinational logic.

In the proposed procedures, there are some common tasks:

1. Inputs (input factors $A \cdot B$ in multiplication or input X in $X(mod\ P)$) are split into subvectors.
2. All subvectors are combined to define a polynomial.
3. This procedure is iterated as long as the result $> 2 \cdot P$.

4.1 Modulo function computation

We propose the following two-step procedure to compute $X(mod\ P)$:

1. X is split into k subvectors with $\leq \delta$ bits in every subvector, where $\delta = \lceil log_2 P - 1 \rceil$.
2. The resulting subvectors are combined according to Eq 3:

$$X(mod\ P) = \sum_{i=1}^{k} X_i \cdot \left(2^{\delta\cdot(i-1)}(mod\ P)\right). \tag{3}$$

This formula can be applied recursively producing reduced intermediate results at every step. The coefficient $2^{\delta\cdot(i-1)}(mod\ P)$ is a constant and it does not exceed $P - 1$. At the first step, it holds that $X_i = 2^\delta - 1$, since Eq. 3 achieves the maximum value. Then Eq. 3 is called recursively until the result is $\leq 2 \cdot P$. At the end, the result is compared with P and, if needed, P is subtracted from the result of the last step. The overall flow (reminiscent of the Fourier computation) is represented by Algorithm 1.

For illustration, consider the following example. Suppose that X is an 18-bit input and $P = 47$. Then modulo P is a 6-bit number, and the input X is split into three 6-bit tuples $X = (X_3, X_2, X_1)$, where $X_1 = (x_6, x_5, \ldots, x_1)$, $X_2 = (x_{12}, x_{11}, \ldots, x_7)$, and $X_3 = (x_{18}, x_{17}, \ldots, x_{13})$. Then $2^6(mod\ 47) = 17(mod\ 47)$ and $2^{12}(mod\ 47) = 7(mod\ 47)$. Hence, in the first iteration Eq. 3 takes the following form:

$$\begin{aligned} X(mod\ 47) &= X_1 + X_2 \cdot 2^6(mod\ 47) + X_3 \cdot 2^{12}(mod\ 47) = \\ &= X_1 + X_2 \cdot 17(mod\ 47) + X_3 \cdot 7(mod\ 47) = S_1 \end{aligned}$$

Algorithm 1 Modulus function computation

Input:
$X = (x_n, x_{n-1}, ..., x_1)$
P

$r = \lceil log_2 P \rceil$
$k = \lceil \frac{n}{r} \rceil$
$length(X)$ - bit range of X
$con(p - q : L)$ - concatenation $(p - q)$ zeros as the most significant bits to L
if $length(X) < k \cdot r$ **then**
    $(k \cdot r - n : X)$
$X = (X^k, X^{k-1}, \cdots, X^1)$
$X^1 = (x_r, x_{r-1}, ..., x_1)$
$X^2 = (x_{2 \cdot r}, x_{2 \cdot (r-1)}, ..., x_{r+1})$
$\cdots$
$X^i = (x_{i \cdot r}, x_{i \cdot (r-1)}, ..., x_{i \cdot (r+1)})$
$\cdots$
$X^k = (x_{k \cdot r}, x_{k \cdot (r-1)}, ..., x_{k \cdot (r+1)})$
$S = \sum_{i=1}^{k} X^i \cdot (2^{i \cdot (r-1)}) (mod P)$
$S_{temp} = S$
while $S_{temp} > 2 \cdot P$ **do**:
    $n_{temp} = length(S_{temp})$
    $k_{temp} = \lceil \frac{n_{temp}}{r} \rceil$
    if $length(S_{temp}) < k_{temp} \cdot r$ **then**
        $(k_{temp} \cdot r - n_{temp} : S_{temp})$
    $S^1_{temp} = (s_r, s_{r-1}, ..., s_1)$
    $S^2_{temp} = (s_{2 \cdot r}, s_{2 \cdot (r-1)}, ..., s_{r+1})$
    $\cdots$
    $S^i_{temp} = (s_{i \cdot r}, s_{i \cdot (r-1)}, ..., s_{i \cdot (r+1)})$
    $\cdots$
    $S^{k_{temp}}_{temp} = (s_{k_{temp} \cdot r}, s_{k_{temp} \cdot (r-1)}, ..., s_{k_{temp} \cdot (r+1)})$
    $S_{temp} = \sum_{i=1}^{k} S^i_{temp} \cdot (2^{i \cdot (r-1)}) (mod P)$
if $P \leq S_{temp}$ **then**
    $S = S_{temp} - P$
else
    $S = S_{temp}$

If input $X = 2^{18} - 1$, then its binary representation requires 18 bits, i.e., $X_1 = X_2 = X_3 = 63_{10} = 111111_2$. Then $S_1 \leq 63 + 63 \cdot 17 + 63 \cdot 7 = 1575_{10} = 11000100111_2$. In this case Eq. 3 takes the following form:

$$S_1(mod\ 47) = S_1^1 + S_2^1 \cdot 2^6(mod\ 47) = \\ = S_1^1 + S_2^1 \cdot 17(mod\ 47) = S_2 \leq 447.$$

If $S_1^1 = 1001110_2$ and $S_2^1 = 11000_2$, it follows $S_2 = 447$. The second iteration splits the 9-bit S_2 number into two 6-bit and 3-bit tuples: $S_2 = \left(S_2^2, S_1^2\right)$, where $S_2^2 = \left(s_9^2, s_8^2, s_7^2\right)$ and $S_1^2 = \left(s_6^2, s_5^2, ..., s_1^2\right)$. In this case Eq. 3 takes the following form:

$$S_2(mod\ 47) = S_1^2 + S_2^2 \cdot 17(mod\ 47) = S_3 \leq 148.$$

If $S_1^2 = 111111_2$ and $S_2^2 = 101_2$, it follows $S_3 = 148$. The third iteration splits the 8-bit number S_3 into two 6-bit and 2-bit tuples: $S_3 = \left(S_2^3, S_1^3\right)$, where $S_2^3 = \left(s_8^3, s_7^3\right)$ and $S_1^3 = \left(s_6^3, s_5^3, ..., s_1^3\right)$. In this case Eq. 3 takes the following form:

$$S_3(mod\ 47) = S_1^3 + S_2^3 \cdot 17(mod\ 47) = S_4 \leq 54.$$

If $S_1^3 = 010100_2$ and $S_2^3 = 10_2$, it follows $S_4 = 54$. Since $S_4 < 2 \cdot P = 94$, S_4 is compared with $P = 47$: if $S_4 > 47$, then $X(mod\ 47) = S_4 - 47$, else $X(mod\ 47) = S_4$.

We provide a Verilog functional representation of this example in Listing 1.1 for $X(mod\, 47)$, where X is a 100−bit number.

Listing 1.1: Modulus function computation

```
module x_100_mod_47(X, S);

input [100:1] X;
output [6:1] S;
wire [14:1] S_temp_1;
wire [11:1] S_temp_2;
wire [9:1] S_temp_3;
wire [8:1] S_temp_4;
wire [7:1] S_temp_5;
reg  [6:1] S_temp;

assign S_temp_1 = X[6:1] + X[12:7]*5'b10001 +
                  X[18:13]*3'b111 + X[24:19]*5'b11001 +
                  X[30:25]*2'b10 + X[36:31]*6'b100010 +
                  X[42:37]*4'b1110 + X[48:43]*2'b11 +
                  X[54:49]*3'b100 + X[60:55]*5'b10101 +
                  X[66:61]*5'b11100 + X[72:67]*3'b110 +
                  X[78:73]*4'b1000 + X[84:79]*6'b101010 +
                  X[90:85]*4'b1001 + X[96:91]*4'b1100 +
                  X[100:97]*5'b10000 ;

assign S_temp_2 = S_temp_1[6:1] +
                  S_temp_1[12:7]*5'b10001 +
                  S_temp_1[14:13]*3'b111 ;
```

```
assign S_temp_3 = S_temp_2[6:1] + S_temp_2[11:7]*5'b10001;

assign S_temp_4 = S_temp_3[6:1] + S_temp_3[9:7]*5'b10001;

assign S_temp_5 = S_temp_4[6:1] + S_temp_4[8:7]*5'b10001;

always @(S_temp_5)
begin
  if (S_temp_5 >= 6'b101111)
    S_temp <= S_temp_5 - 6'b101111;
  else
    S_temp <= S_temp_5;
end

assign S = S_temp;

endmodule
```

4.2 Computation of the modular product

We propose the following two-step procedure to compute the product $A \cdot B = R(mod\ P)$, where $A = (A_\delta, A_{\delta-1}, ..., A_1)$, $B = (B_\delta, B_{\delta-1}, ..., B_1)$, and the δ-subvectors A_δ and B_δ consist of the most significant bits. For example, if A and B are 12-bit numbers and $\delta = 4$, then $A_4 = (a_{12}, a_{11}, a_{10})$ and $B_4 = (b_{12}, b_{11}, b_{10})$, where a_{12} and b_{12} are the most significant bits.

This contribution proposes a modulus function computation for an arbitrary modulo without limitation on the value of P. The idea of the approach is to use a large set of small moduli instead than a small set of large moduli, as it is used traditionally. Hence we consider that A, B and P vary from 6 to 12 bits.

1. The inputs are split into 2-, 3- and 4-bit subvectors.
2. The corresponding pairs of subvectors are multiplied applying the following recursive formula:

$$R = \sum_{i=1}^{\delta} \sum_{j=1}^{\delta} A_i \cdot B_j \cdot \left(2^{m \cdot (i+j-2) \cdot 3} (mod\ P)\right) = S\_temp. \tag{4}$$

The maximum value of S_temp does not exceed $2^{3 \cdot \delta + 2}$, $2^{3 \cdot \delta + 3}$ or $2^{3 \cdot \delta + 4}$ depending on the value of modulo P.

As an illustration, consider three common cases:

1. $\delta = 2$, then $S\_temp \leq 2^8$ and $S\_temp_2 = S\_temp[3:1] + S\_temp[6:4] \cdot 2^3 (mod\ P) + S\_temp[8:7] \cdot 2^6 (mod\ P)$;
2. $\delta = 3$, then $S\_temp \leq 2^{12}$ and $S\_temp_2 = S\_temp[3:1] + S\_temp[6:4] \cdot 2^3 (mod\ P) + S\_temp[9:7] \cdot 2^6 (mod\ P) + S\_temp[12:10] \cdot 2^9 (mod\ P)$;
3. $\delta = 4$, then $S\_temp \leq 2^{12}$ and $S\_temp_2 = S\_temp[3:1] + S\_temp[6:4] \cdot 2^3 (mod\ P) + S\_temp[9:7] \cdot 2^6 (mod\ P) + S\_temp[12:10] \cdot 2^9 (mod\ P) + S\_temp[15:13] \cdot 2^{12} (mod\ P)$.

Finally, if $S\_temp_2 > P$, then $S = S\_temp_2 - P$, otherwise $S = S\_temp_2$.

Let us multiply the two 6-bits numbers A and B as $A \cdot B = S(mod\ 47)$. Splitting the operands into two (i.e., $\delta = 2$) 3-bits subvectors, Eq. 4 becomes:
$A \cdot B = S(mod\ 47) = A_1 \cdot B_1(mod\ 47) + A_1 \cdot B_2 \cdot 2^3(mod\ 47) + A_2 \cdot B_1 \cdot 2^3(mod\ 47) + A_2 \cdot B_2 \cdot 2^6(mod\ 47) = S\_temp$.

When $A = 45$ and $B = 15$, S_temp achieves the maximum value, which is $158_{10} = 10011110_2$: $A_1 = 101_2$, $A_2 = 101_2$, $B_1 = 111_2$, $B_2 = 1_2$, hence $A \cdot B = 5 \cdot 7(mod\ 47) + 5 \cdot 1 \cdot 2^3(mod\ 47) + 5 \cdot 7 \cdot 2^3(mod\ 47) + 5 \cdot 1 \cdot 2^6(mod\ 47) = 35(mod\ 47) + 40(mod\ 47) + 45(mod\ 47) + 38(mod\ 47) = 158$. Trying another value for A and B, it is $S\_temp < 158$.

The second iteration reduces S_temp to a value < 47. Assume that $S\_temp = 158$, then $S\_temp_2 = 6 + 3 \cdot 2^3(mod\ 47) + 2 \cdot 2^6(mod\ 47) = 6 + 24 + 34 = 64$.

Finally, taking into account that $64 > 47$, the result is $S = 64 - 47 = 17$. Note that the bit range of S_temp is preselected.

We provide a Verilog functional representation of this example in Listing 1.2 for the modular multiplication of two 6-bit operands by modulo 47.

Listing 1.2: Modular multiplication $A \cdot B(mod\ 47)$

```
module mult_mod_47_bits(A, B, R);

input  [6:1] A, B;
output [6:1] R;
wire   [6:1] r1,r2,r3,r4,r5,r6;
wire   [8:1] temp_R_1;
wire   [7:1] temp_R_2;
reg    [6:1] temp_R;

mult_3x3 label1 (.a1(A[3]),.a2(A[2]),.a3(A[1]),
                 .b1(B[3]),.b2(B[2]),.b3(B[1]),
                 .r1(r1[6]),.r2(r1[5]),.r3(r1[4]),
                 .r4(r1[3]),.r5(r1[2]),.r6(r1[1]));

mult_3x3_8 label2 (.a1(A[3]),.a2(A[2]),.a3(A[1]),
                   .b1(B[6]),.b2(B[5]),.b3(B[4]),
                   .r1(r2[6]),.r2(r2[5]),.r3(r2[4]),
                   .r4(r2[3]),.r5(r2[2]),.r6(r2[1]));

mult_3x3_8 label3 (.a1(A[6]),.a2(A[5]),.a3(A[4]),
                   .b1(B[3]),.b2(B[2]),.b3(B[1]),
                   .r1(r3[6]),.r2(r3[5]),.r3(r3[4]),
                   .r4(r3[3]),.r5(r3[2]),.r6(r3[1]));

mult_3x3_17 label4 (.a1(A[6]),.a2(A[5]),.a3(A[4]),
                    .b1(B[6]),.b2(B[5]),.b3(B[4]),
                    .r1(r4[6]),.r2(r4[5]),.r3(r4[4]),
                    .r4(r4[3]),.r5(r4[2]),.r6(r4[1]));

assign temp_R_1 = r1 + r2 + r3 + r4;

mult_3_8 label5 (.a1(temp_R_1[6]),.a2(temp_R_1[5]),
                 .a3(temp_R_1[4]),
```

```
                    .r1(r5[6]),.r2(r5[5]),.r3(r5[4]),
                    .r4(r5[3]),.r5(r5[2]),.r6(r5[1]));

mult_2_17 label6 (.a1(temp_R_1[8]),.a2(temp_R_1[7]),
                  .r1(r6[6]),.r2(r6[5]),.r3(r6[4]),
                  .r4(r6[3]),.r5(r6[2]),.r6(r6[1]));

assign temp_R_2 = temp_R_1[3:1] + r5 + r6;

always @(temp_R_2)
begin
  if (temp_R_2 >= 47)
    temp_R <= temp_R_2 - 47;
  else
    temp_R <= temp_R_2;
end

assign R = temp_R;

endmodule
```

There are six external blocks in Listing 1.2: *mult_*3x3, two *mult_*3x3_8, *mult_*3x3_17, *mult_*3_8, and *mult_*2_17. All these blocks consist of Boolean functions with hundreds of product terms (which we do not represent to save space). In the next section we discuss the impact of Boolean function minimization in modular computations.

5 Boolean minimization in modular operations

The result of any arithmetic computation can be represented as 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.

For a pair of δ-bit tuples, consider $2^i (mod\ P)\ X_i \cdot \left(2^{\delta\cdot(i-1)} (mod\ P)\right.$, where $i = 1, 2, \ldots, k$, are the corresponding factors of the multiplication. Then $2^i (mod\ P)$ is a constant whose bits are redundant in the minimization, because all rows in the truth table corresponding to this constant have the same value $2^i (mod\ P)$.

The initial truth table for $X (mod\ P)$ consists of P rows and $2 \cdot \delta$ columns, where the left δ columns correspond to all integers from 0 up to $P-1$, and the right columns correspond to $X \cdot 2^i (mod\ P)$.

Example Consider $2^8 (mod\ 13) = 9 (mod\ 13) = 1001_2$. In this case, subtable 1 represents the truth table for $X \cdot 9 (mod\ 13)$ before minimization and subtable 2 represents the SOP after minimization (it can obtained by tools like [12] or ELS [13]). So the first four bits in the last row of the truth table in subtable 1 represent 12_{10}, and the right four bits represent $12 \cdot 9 (mod\ 13) = 4_{10}$. For the 18-bit input X and $P = 47$, all pairs of corresponding factors are represented as a SOP: with 12 columns (6 inputs and 6 outputs) $X_2 \cdot 17 (mod\ 47)$ and $X_3 \cdot 7 (mod\ 47)$; with 11 columns (5 inputs and 6 outputs) $X_2^1 \cdot 17 (mod\ 47)$; with 9 columns (3 inputs and 6 outputs) $X_2^2 \cdot 17 (mod\ 47)$; with 8 columns (2 inputs and 6 outputs) $X_2^3 \cdot 17 (mod\ 47)$.

Table 1: Representation of $X(mod\ 13)$ with SOPs

$x_4x_3x_2x_1$	$r_4r_3r_2r_1$	$x_4x_3x_2x_1$	$r_4r_3r_2r_1$
0 0 0 1	0 0 0 0	0 1 0 0	1 0 0 0
0 0 0 0	1 0 0 1	1 0 1 -	1 0 0 0
0 0 1 0	0 1 0 1	0 0 0 1	1 0 0 0
0 0 1 1	0 0 0 1	0 1 1 1	1 0 0 0
0 1 0 0	1 0 1 0	0 1 0 1	0 1 0 0
0 1 0 1	0 1 1 0	- 0 1 0	0 1 0 0
0 1 1 0	0 0 1 0	1 - 0 0	0 1 0 0
0 1 1 1	1 0 1 1	1 0 0 -	0 0 1 0
1 0 0 0	0 1 1 1	0 1 - -	0 0 1 0
1 0 0 1	0 0 1 1	- 0 0 1	0 0 0 1
1 0 1 0	1 1 0 0	1 0 0 -	0 0 0 1
1 0 1 1	1 0 0 0	0 - 1 1	0 0 0 1
1 1 0 0	0 1 0 0	0 0 1 -	0 0 0 1
Subtable 1		Subtable 2	

In the example from the last section in Listing 1.2, modular multiplication of two 6-bit numbers by modulo 47 includes six blocks. Every block realizes Boolean functions minimized with Espresso or ELS:

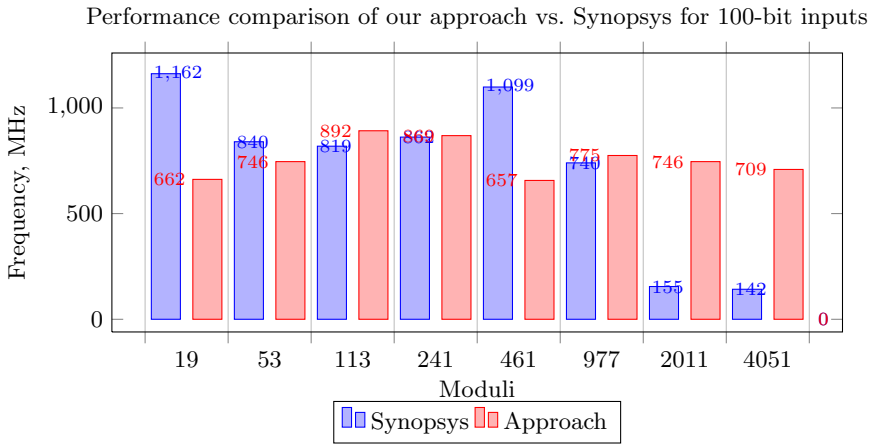
- the block *mult_*3x3 realizes 3 by 3 bit multiplication, with 6 inputs and 6 outputs;
- the block *mult_*3x3_8 realizes multi-operand multiplication by modulo 47: $([3:1]\cdot[3:1]\cdot 8)(mod\ 47)$, with 6 inputs and 6 outputs;
- the block *mult_*3x3_17 realizes multi-operand multiplication by modulo 47: $([3:1]\cdot[3:1]\cdot 17)(mod\ 47)$, with 6 inputs and 6 outputs;
- the block *mult_*3_8 realizes multi-operand multiplication by modulo 47: $([3:1]\cdot 8)(mod\ 47)$, with 3 inputs and 6 outputs, and the three least significant bits equal to zero;
- the block *mult_*2_17 realizes multi-operand multiplication by modulo 47: $([2:1]\cdot 17)(mod\ 47)$, with 2 inputs and 6 outputs, and the fourth bit equal to zero.

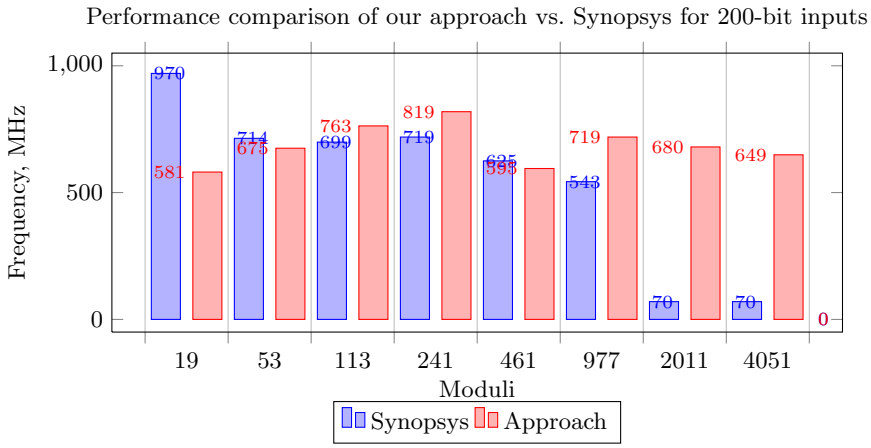
6 Experimental results

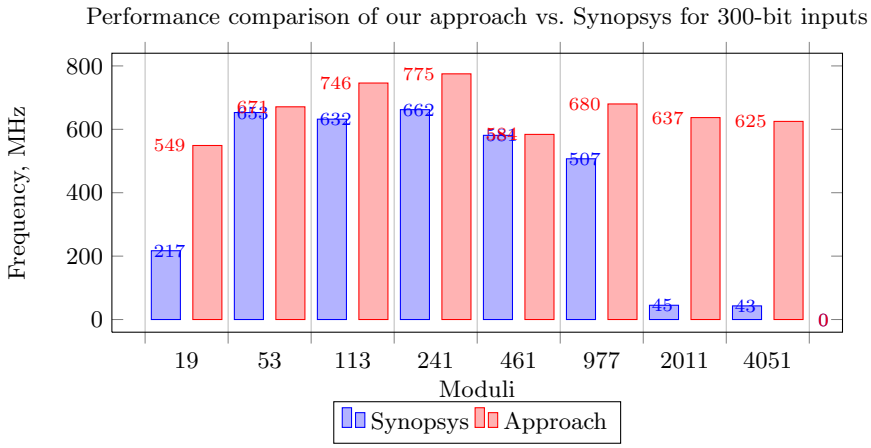
We compared our procedure vs. three EDA tools: Synopsys, Mentor Graphics (for standard cells), and Xilinx (for FPGAs). Since Mentor Graphics and Xilinx do not synthesize general modular operations, we compared with the case of special moduli, such as $2^s-1, 2^s+1$. Our approach shows gains within 10%.

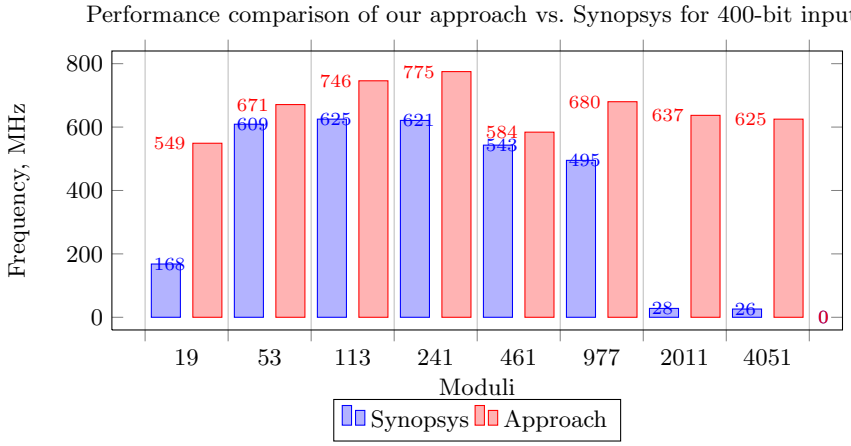
Synopsys is the only EDA tool which generates $X(mod\ P)$ circuits. We report results of synthesis using Synopsys 2014 on 28 nm Standard Cell ASIC technology from United Microelectronics Corporation.

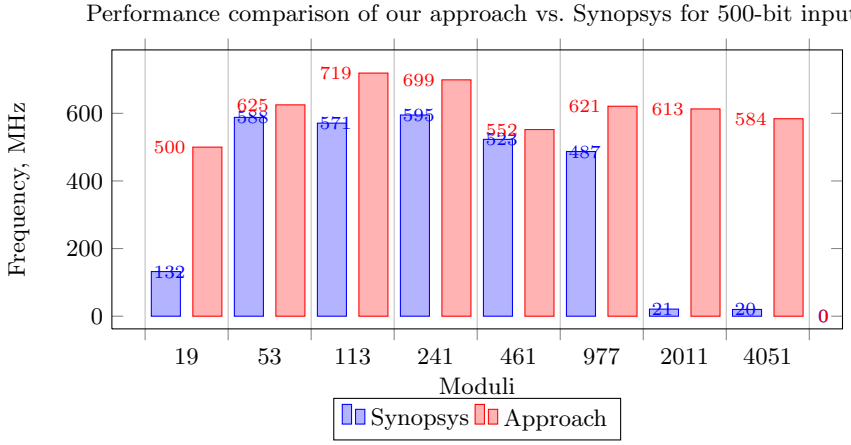
The first five plots compare the latency of circuits of $X(mod\ P)$ (in MHz) for inputs X of 100, 200, 300, 400, and 500 bits, respectively, for moduli P of 19 (5 bits), 53 (6 bits), 113 (7 bits), 241 (8 bits), 461 (9 bits), 997 (10 bits), 2011 (11 bits), and 4051 (12 bits).


Performance comparison of our approach vs. Synopsys for 100-bit inputs
Frequency, MHz
1,000
500
0
1,162
662
840
746
892
819
862
862
1,099
657
775
740
746
155
709
142
0
19
53
113
241
461
977
2011
4051
Moduli
Synopsys
Approach


Performance comparison of our approach vs. Synopsys for 200-bit inputs
Frequency, MHz
1,000
500
0
970
581
714
675
763
699
819
719
635
593
719
543
680
70
649
70
0
19
53
113
241
461
977
2011
4051
Moduli
Synopsys
Approach


Performance comparison of our approach vs. Synopsys for 300-bit inputs
Frequency, MHz
800
600
400
200
0
549
217
671
653
746
632
775
662
581
581
680
507
637
45
625
43
0
19
53
113
241
461
977
2011
4051
Moduli
Synopsys
Approach



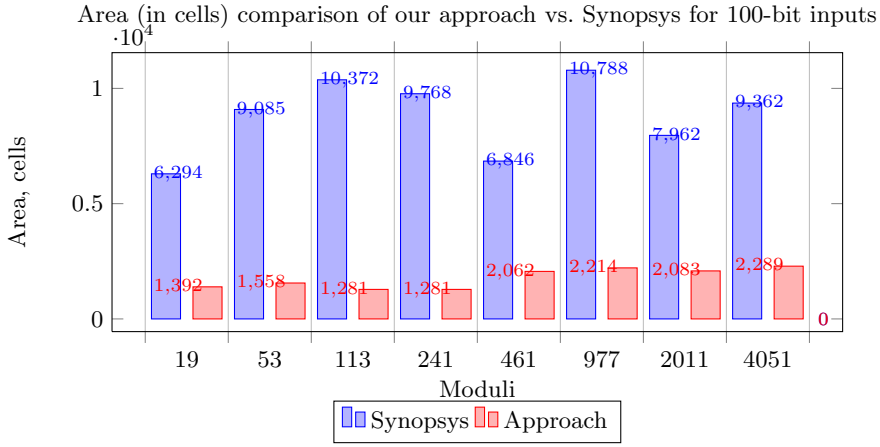


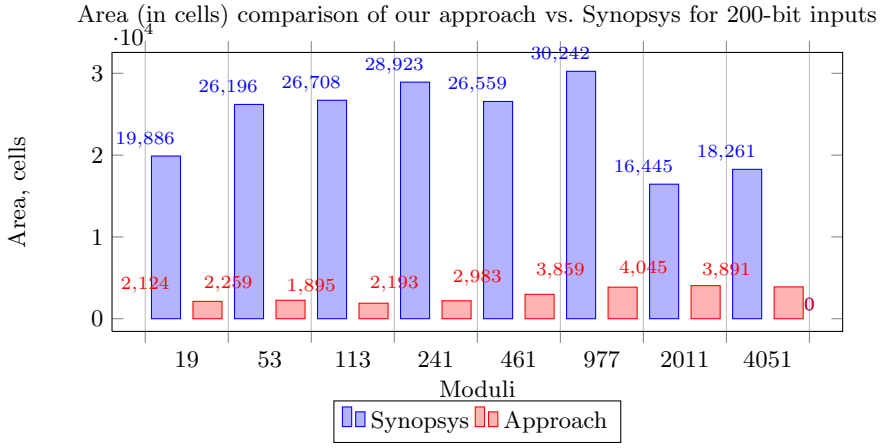
The performance results are grouped into three categories:

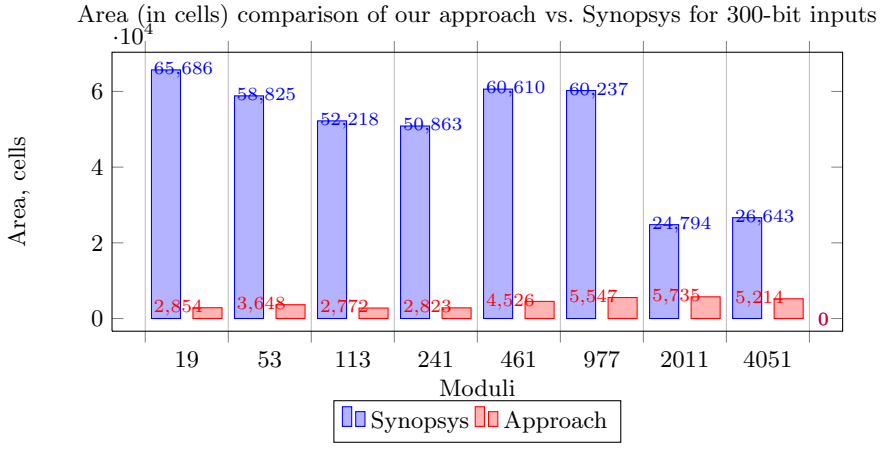
1. The proposed approach is faster by dozens of times (up to 30 times) for 11- and 12-bit moduli for all ranges of inputs (two right parts of the plots);
2. The proposed approach is faster on average by 15% (up to 300 %) for almost all 5-bit to 10-bit moduli for all ranges of inputs (central and left parts of the plots);
3. The proposed approach is slower, with a significant difference only for three cases (for 5-bit moduli and 100- and 200-bit inputs, and for 9-bit moduli and 10-bit input - 75%, 66%, and 67%, respectively), and a minor advantage (up to 5% on average) for 6-bit and 7-bit moduli for 100-bit input, for 6-bit and 9-bit moduli for 200-bit input.

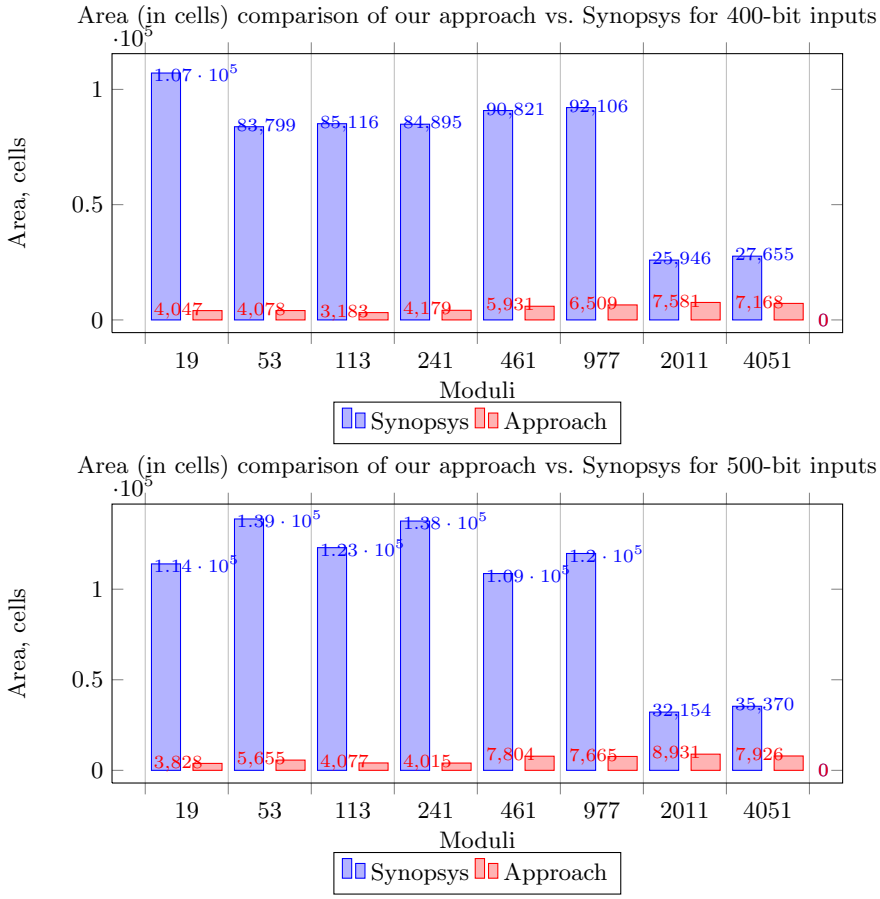
Note that the experiments have been conducted for 40 different values.

The next five plots compare the area of circuits of $X(mod\ P)$ (number of cells from the library *cells*) for inputs X of 100, 200, 300, 400, and 500 bits, respectively, using the same moduli as in the experiments for performance.


Area (in cells) comparison of our approach vs. Synopsys for 100-bit inputs
$\cdot 10^4$
Area, cells
1
0.5
0
6,294
1,392
9,085
1,558
10,372
1,281
9,768
1,281
6,846
2,062
10,788
2,214
7,962
2,083
9,362
2,289
0
19
53
113
241
461
977
2011
4051
Moduli
Synopsys
Approach


Area (in cells) comparison of our approach vs. Synopsys for 200-bit inputs
$\cdot 10^4$
Area, cells
3
2
1
0
19,886
2,124
26,196
2,259
26,708
1,895
28,923
2,193
26,559
2,983
30,242
3,859
16,445
4,045
18,261
3,891
0
19
53
113
241
461
977
2011
4051
Moduli
Synopsys
Approach


Area (in cells) comparison of our approach vs. Synopsys for 300-bit inputs
$\cdot 10^4$
Area, cells
6
4
2
0
65,686
2,854
58,825
3,648
52,218
2,772
50,863
2,823
60,610
4,526
60,237
5,547
24,794
5,735
26,643
5,214
0
19
53
113
241
461
977
2011
4051
Moduli
Synopsys
Approach



The area results show the advantage of the proposed approach which improves up to 30 times for all moduli and input values.

7 Conclusions and further research

Performance of computer arithmetic is one of the main advantages of RNS vs. traditional approaches. We proposed a technique that improves significantly area and performance of RNS vs. synthesis using standard EDA tools.

The experiments show significant gains by our approach vs. Synopsys. The gain in performance is up to 30 times and in area is up to 15 times. Moreover, Synopsys could not synthesize circuits for inputs X larger than 500 bits: the synthesis by Synopsys of the modulo function for a 600-bit input X failed after nine days, whereas it takes only 20 minutes with our approach.

Our approach is not limited to modular multiplication and to the modulo function, but it can be extended to any arithmetic operation. Dozens of circuits were designed with the technique presented here and then embedded in arithmetic units by the hi-tech factory Integral (Minsk, Belarus).

Topics of further research include:

1. Comparing different forms of representations (SOPs, Reed-Muller expansions, binary decision diagrams) in the realization of partial products;
2. Designing FPGAs using Xilinx and Altera architectures.

References

1. N.I Cherviakov et al., "Modular Structures of Parallel Computing Systems for Neuroprocessors", Moscow, Russia, 2003, 288 p. (in Russian).
2. L. Sousa and R. Chaves, "A Universal Architecture for Designing Efficient Modulo $2^n \pm 1$ Multipliers", IEEE Transactions on Circuits and Systems, 2005, Vol. 52, 6, p. 1166-1178.
3. R. Zimmermann, "Efficient VLSI Implementation of Modulo $2^n + 1$ Addition and Multiplication", 14th IEEE Symposium on Computer Arithmetic, Adelaide, Australia, Apr. 14-16, 1999, p. 158-167.
4. B.M. Malashevich, " Unknown Modular Supercomputers," Proceedings of Conference for 50 years of modular arithmetic, Nov. 23-25, 2005, pp. 50-70 (in Russian).
5. 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.
6. E. Ozturk, B. Sunar, E. Savas, "Low-Power Elliptic Curve Cryptography Using Scaled Modular Arithmetic", Proceedings of the 6th International Workshop Cryptographic Hardware in Embedded Systems, Cambridge, MA, USA, Aug. 11-13, 2004, Vol. 3156, p. 92-106.
7. P.L. Montgomery, "Modular Multiplication without Trial Division Mathematics of Computation", Mathematics of Computation, Vol. 44, No. 170. Apr., 1985, p. 519-521.
8. "Computers, Software, Engineering and Digital Devices", Ed. R.C. Dorf. Taylor and Francis, 2006, 576 p.
9. P.V.A. Mohan, "Residue Number Systems. Theory and Applications", Birkhäuser (Springer International Publishing), 2016.
10. A.R. Omondi and B. Premkumar, Residue Number System: Theory and Implementation, Imperial College Press, 2007.
11. J.T. Butler and T. Sasao, "Fast hardware computation of x mod z", 25th IEEE International Parallel and Distributed Processing Symposium Anchorage, Ak, USA, May 16-17, 2011, p. 289-292.
12. https://embedded.eecs.berkeley.edu/pubs/downloads/espresso/index.htm
13. 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).
14. V.P. Irhin, "Tabular implementation of modular arithmetic operations" // Sb.nauch.tr. YUbilejnoj Mezhdunarodnoj nauchno-tekhnicheskoj konferencii 50 let modulyarnoj arifmetiki, 2005. pp. 268-273 (in Russian).
15. D. Gorodecky, "Hardware realization of X (mod P)", Proceedings of the International scientific-practical conference "Actual problems of Radio Physics", Tomsk, Russian Federation, Oct. 3-6, 2013. Izvestia VUZov. Physic. Vol. 56. 9/2. P. 198-199.
16. M.A. Will and Ryan K. L. Ko, "Computing Mod Without Mod"

17. S.J. Piestrak, "Design of residue generators and multi-operand modulo adders using carry save adders", Proceedings of the 10th Symposium on Computer Arithmetic, Grenoble, 2628 June 1991. pp. 100107.
18. P.V.A. Mohan, Efficient design of binary to RNS converters. J. Circuit. Syst. Comp 9, 1999, pp. 145154.
19. A.B. Premkumar, "A formal framework for conversion from binary to residue numbers" // IEEE Trans. Circuits Syst., 2002, vol. 49, pp. 135144.
20. A.B. Premkumar, E.L. Ang, E.M.K. Lai, "Improved memory-less RNS forward converter based on periodicity of residues" // IEEE Trans. Circuits Syst., 2006, vol. 53, pp. 133137.
21. J.Y.S. Low, C.H. Chang, "A new approach to the design of efficient residue generators for arbitrary moduli" // IEEE Trans. Circuits Syst. I Reg. Papers 60, 2013, pp. 23662374.
22. http://www.computer-museum.ru/histussr/sokconf0.htm
23. C. Ding, D. Pei, T.Pei, A. Salomaa, "Chinese Remainder Theorem: Applications in Computing, Coding, Cryptography" // Harbin Institute of Technology Press, 1996.