Method and device to implement accumulating to arbitrary modulo in cryptographic applications

V. I. Petrenko, V. V. Kopytov, M. P. Sutormin

***Abstract*—This article examines the methods of accumulating in the context of operations of multiplying numbers to arbitrary modulo in cryptographic applications. The authors present a new approach to data processing using methods accumulating to arbitrary modulo, which when encrypting information will improve the performance of cryptographic transformation operations. The article describes in detail the main methods of accumulating to arbitrary modulo, their advantages and potential uses in cryptographic applications. The research, methodologies and practical solutions presented in the article are of interest to specialists in the field of cybersecurity, as well as to developers of high-tech software and IS software and hardware.**

***Keywords*—modulo addition, accumulating, cybersecurity, hardware description language, asynchronous encryption.**

I. Introduction

When exchanging information between various objects of informatization and information technologies, one of the main tasks is to ensure the confidentiality of information that has intellectual or economic value for the owner of the information and is not common for general use. This problem is currently solved by various methods, the most reliable of which is the method of cryptographic transformations during information exchange [1]. For such purposes, standard symmetric and asymmetric encryption algorithms are used. Asymmetric algorithms are more complex to calculate and implement than symmetric ones [2].

The RSA (Rivest-Shamir-Adleman) algorithm [4], [5] is one of the most widely used public key encryption algorithms, which is based on the problems of generating prime numbers, multiplying numbers modulo and decomposing composite numbers into prime factors.

Due to the popularity and widespread use of the RSA algorithm, a number of elementary attacks are known that hinder its use for small cryptographic key lengths [6], [7].

To counter such attacks, the length of the RSA cryptographic key may be increased, which complicates its implementation and leads to a decrease in its speed [8].

Article received on May 17, 2024.

Petrenko Vyacheslav Ivanovich, Head of a Department of the Institute of Digital Development of the North Caucasus Federal University, Stavropol, Russia (email: vipetrenko@ncfu.ru).

Kopytov Vladimir Vyacheslavovich, Professor of the Institute of Digital Development of the North Caucasus Federal University, Stavropol, Russia (email: vkopytov@ncfu.ru).

Sutormin Matvey Pavlovich, student of the “Information security” Training program of the North Caucasus Federal University, Stavropol, Russia (email: sutorminp@gmail.com).

The greatest complexity in the implementation of the algorithm comes from the operation of multiplying numbers modulo other than 2^n [9]. Reducing the complexity of implementing the operation of multiplying numbers to arbitrary modulo is possible by reducing intermediate partial products obtained as a result of arbitrary modulo multiplication, followed by the operation of accumulating these partial products to the same arbitrary modulus. The implementation of this method with hardware solutions will further improve the performance of cryptographic transformation operations. Therefore, the development of an effective method of accumulating is an urgent task to increase the performance of cryptographic algorithms.

II. Application of modulo addition in the RSA algorithm

The RSA algorithm uses modulo addition to encrypt and decrypt data using public and private keys, respectively, as part of an arbitrary modulo multiplication operation.

When encrypting with a public key, the sender takes the following steps:

1. Selects the recipient's public key, which consists of two numbers: (e, p), where p is the product of two prime numbers b and q, and e is the public key.
2. Converts the original message M to an integer m, where $m<p$.
3. Calculates an encrypted message c using the public key according to the formula:

$$c = m^e \bmod p, \quad (1)$$

where c is the encrypted message, m is the plaintext, e is the public key, p is the modulus.

4. Sends the encrypted message c.

When decrypting using a private key, the recipient completes the following steps:

1. Receives the encrypted message c.
2. Selects a private key, which consists of two numbers: (d, p), where d is the private key.
3. Calculates the integer m using the private key according to the formula:

$$m = c^d \bmod p. \quad (2)$$

4. Converts the integer m to the original message M.

When implementing this algorithm, the main operation is the operation of raising numbers to a power modulo the corresponding one. Raising a number to a power can be thought of as repeatedly multiplying a number by itself and then modulo the result. For example, to raise number a to the power of n modulo p, one needs to multiply a by itself (n−1) times, and then convert the result modulo p. In practice, a modulo reduction operation is usually performed after each

multiplication operation. Faster algorithms for raising numbers to powers are also used, which involve expanding the power and performing the corresponding transformations. When implementing such algorithms, the operation of accumulating modulo p is used as part of the operations of exponentiation, which involves finding the remainder when dividing the sum of numbers by p. That is, if the sum exceeds p, the remainder of the sum divided by p is taken so that it remains in the range from 0 to (p−1). Using these two concepts, one can effectively represent the operation of exponentiation as a sequence of addition and remainder operations, which can be useful, for example, when working with large numbers used in cryptographic applications.

III. Accumulating methods

The classic method of constructing modulo 2^n accumulators, where n is the number capacity, is to add a register to the accumulator output and create feedback from the register output to the second information inputs of the accumulator. The accumulating principle is that the accumulator has one input to which a sequence of numbers A_i is supplied and these numbers are sequentially summed modulo 2^n, forming the sum S_i:

$$S_i = (A_i + S_{i-1})\ mod\ 2^n. \quad (3)$$

These schemes are widely known and their implementation is presented in [10]. However, in a number of applications, including cryptographic ones, problems of accumulating modulo other than 2^n often arise, for example:

$$2^n \pm k, where\ 2^{n-1} > k \geq 1. \quad (4)$$

There are known methods for n-bit arbitrary modulo addition, used in a modulo accumulator [11], [12]. The main idea of these methods is that the integer numbers A_i, (i=1, 2, 3...), $0 \leq A_i < p$, arriving at the input of the accumulator, are summed clock by clock with the numbers S_{i-1}, written in its memory at the previous clock cycle. The result of the addition $A_i + S_{i-1}$ is taken modulo p as follows. If $(A_i + S_{i-1}) < p$, then the usual addition $(A_i + S_{i-1})$ is performed and this sum is the result of S_i. If $(A_i + S_{i-1}) \geq p$, then the p value is subtracted from the sum $(A_i + S_{i-1})$ and the result S_i is the sum $(A_i + S_{i-1})$ mod p. The result is written to the device's memory and used as the value of the number S_{i-1} at the next clock cycle.

$$S_i \equiv (A_i + S_{i-1})\ mod\ p,\ (i = 1, 2, 3\ ...). \quad (5)$$

Moreover, S_i is calculated in the following sequence:

$$S_i' = A_i + S_{i-1}, \quad (6)$$

$$S_i'' = S_i' - p, \quad (7)$$

$$S_i = \begin{cases} S_i', if\ (A_i + S_{i-1}) < p \\ S_i'', if\ (A_i + S_{i-1}) \geq p \end{cases}, \quad (8)$$

where S_i is the value of the sum modulo p at the i clock cycle, $S_0 = 0$.

This method in [11] is implemented by sequential calculation of S_i' and S_i'' by two different adders. The determination of S_i is carried out based on the results of calculating S_i' and S_i'' in accordance with (8). This implementation, firstly, increases the time for generating the result, and secondly, leads to inefficient use of equipment, since only one adder operates at a time.

The works [12], [13], [14], [15], [16] propose options for parallel calculations of S_i' and S_i'', but this complicates the structure of the adder that implements the operation of finding S_i'', since it must be implemented as a three-input adder.

Thus, the known hardware implementations of the arbitrary modulo accumulating operation either use the equipment inefficiently and have a long addition time, or have a complex implementation of three-input adders, which limits their use when adding sequences with a large amount of numbers and large bit capacity.

In order to increase the efficiency of equipment use and improve the speed of adding large sequences of numbers, a method of accumulating addition to arbitrary modulo is proposed, which helps divide the addition of a sequence of numbers into two streams and effectively use the equipment. The essence of this method is as follows.

The input sequence of non-negative integer numbers $A_i\ (i = 1, 2, 3, \dots), 0 \leq A_i < p_k, k = 1, 2$ is summed clock by clock in two separate streams with the numbers written in its memory on previous clock cycles, forming two independent output sequences of numbers $S_{k,i}$, associated respectively with odd and even clock numbers. Odd and even output sequences are generated alternately. Addition for odd and even number streams can be carried out for different moduli p_1 и p_2, respectively. Let us denote the modulo sum for the first (odd) stream of numbers as $S_{1,i}$ and for the second (even) stream of numbers as $S_{2,i}$. Then:

$$S_{1,i-1} = (\sum A_i)\ mod\ p_1, (i = 1, 3, 5, \dots), \quad (9)$$
$$S_{2,i-1} = (\sum A_i)\ mod\ p_2, (i = 2, 4, 6, \dots). \quad (10)$$

To implement the proposed method of accumulating, it is necessary to add one more memory cell to the known implementation options [11].

One memory cell at each clock cycle stores input numbers A_i with n-bit capacity and output numbers $S_{1,i}$ at the even clock cycle and $S_{2,i}$ at the odd clock cycle with n-bit width.

The second memory cell stores the sum $(A_i + S_{1,i})$ at the even clock cycle and the sum $(A_i + S_{2,i})$ at the odd clock cycle during the next clock cycle. Next, the sum $(A_i + S_{k,i})$ is reduced modulo p_k, where k=1, 2. If the specified sum is greater than modulo p_k, then this modulus is subtracted from it, otherwise the sum without change is sent to the output of the device.

IV. Algorithm for accumulating to arbitrary modulo

The proposed method of accumulating can be implemented by the following algorithm, which uses the following notation: A_i is an input sequence of non-negative integer numbers, consisting of two independent streams of numbers, alternating for even i and odd i, integer non-negative moduli p_1 and p_2, over which accumulating is carried out, respectively, for the first and second stream of numbers, and the values of A_i do not exceed the values of the corresponding moduli p_1 and p_2, $S_{1,i}$ is accumulating sum modulo p_1, $S_{2,i}$ is accumulating sum modulo p_2.

1) Start.
2) Input M.

3) Assign $i = 1$; $k = 1$; $S_{1,0} = 0$, $S_{2,0} = 0$.
4) Input A_i.
5) Assign $S'_{k,i} = A_i + S_{k,i-1}$.
6) Assign $i = i + 1$.
7) Assign $S''_{k,i} = S'_{k,i-1} - P_k$.
8) If $S''_{k,i} < 0$, then assign $S_{k,i} = S''_{k,i}$, otherwise assign $S_{k,i} = S'_{k,i-1}$.
9) Output $S_{k,i}$.
10) If $i > M$, then go to step 13.
11) Assign $k = 2 - i \bmod 2$.
12) Go to step 4.
13) End.

V. Implementation of the Accumulating Algorithm in the Verilog HDL Hardware Description Language

The proposed method of accumulating to arbitrary modulo can be implemented using a hardware description language, which is used for modeling and designing digital systems [17], [18]. An accumulator with two channels for n=8 numbers, implementing the parallel addition method, performed in Verilog HDL [19], is presented in Fig. 1.

The software module consists of blocks describing the inputs and outputs of the modulo accumulator itself, declaring variables and describing the inputs and outputs of two registers, two adders and a multiplexer. The software module uses libraries implemented in a standard way that describe adders, registers and a multiplexer. First, libraries with adders ("adder.v"), registers ("registrs.v"), multiplexer ("mux.v"), in which the corresponding moduli are described, are imported. The implementation of such libraries is standard and is not shown in this listing. The "modulo_adder" specifies the inputs (input_a, clk, P) and outputs (mux_out) of the device. The 8-bit wire connections are then declared, which are described in more detail in the code comments.

The following snippet creates objects and configures the device's inputs and outputs:

– reg_16_bit is a 16-bit register, input_a is the first information input of the register, input_b is the second information input of the register, clk is the clock input and corresponding outputs out_a, out_b;
– adder_8 is the 8-bit adder;
– a and b are adder inputs, out and carry are outputs;
– reg9Bit is the 9-bit register;
– data_in is the data input;
– clk is the clock input and two outputs data_out_8bit (8-bit), data_out_1bit (1-bit);
– adder_9 is the 9-bit adder;
– a and b are 9-bit adder inputs and outputs out, cin, carry;
– multiplexer2to1 is the multiplexer,
– input_from_sum is the input connected to the adder;
– input_from_reg is the input connected to the register, control is the control input;
– out is the information output.

To check the correct operation of the accumulating module, a testing module was developed, the code of which is presented in Fig. 2. At the beginning of this module, the modulo accumulator is imported.

```
`include "adder.v"
`include "registrs.v"
`include "mux.v"
module modulo_adder (
  // inputs
  input [7:0] input_a,
  input clk,
  input [7:0] P,
  // outputs
  output [7:0] mux_out
);
  // internal wires
  wire [7:0] reg16_out_a; // lower 8 digits of the 16-bit register
  wire [7:0] reg16_out_b; // higher 8 digits of the 16-bit register
  wire [7:0] sum1_out;  // 8-bit output of the first 8-bit adder
  wire sum1_carry;  // carry output of the first 8-bit adder
  wire [7:0] reg9_out_8;  //8 bit output of 9-bit register
  wire reg9_out_carry; // output of the 9th bit of the 9-bit register
  wire [7:0] sum2_out; // 8 bit output of the second adder (which performs subtraction)
  wire sum2_out_carry; // carry output of the second adder (which performs subtraction)
  // 16-bit register
  Reg16Bit reg_16_bit (
     .input_a(input_a),
     .input_b(mux_out),
     .clk(clk),
     .out_a(reg16_out_a),
     .out_b(reg16_out_b)
  );
  // 8-bit adder
  Adder_8 adder_8(
     .a(reg16_out_a),
     .b(reg16_out_b),
     .out(sum1_out),
     .carry(sum1_carry)
  );
  // 9-bit register
  Reg9Bit reg9Bit(
     .data_in({sum1_carry, sum1_out}),
     .clk(clk),
     .data_out_8bit(reg9_out_8),
     .data_out_1bit(reg9_out_carry)
  );
  // 9-bit adder
  Adder_9 Adder_9(
     .a({reg9_out_carry, reg9_out_8}),
     .b({1'b1, P}),
     .cin({1'b1}),
     .out(sum2_out),
     .carry(sum2_out_carry)
  );
  // 8-bit multiplexer
  Multiplexer2to1 Multiplexer2to1(
     .input_from_sum(sum2_out),
     .input_from_reg9(reg9_out_8),
     .control(sum2_out_carry),
     .out(mux_out)
  );
```

Fig. 1 – Modulo accumulator in Verilog HDL for number bit width n=8

Next, the variables are created: clk is the clock signal, input_a is the 8-bit input of the device, P_reverse is the input for the inverse code of the module, mux_out is the output of the multiplexer. Then the tb(testbench) module is created, in which testing will take place. In the next block, a modulo_adder object is created, which was shown in Fig. 1, and its inputs and outputs are configured. Next, the period for the clock signal and a description of the alternation of the reverse module P_reverse are introduced. The initialization of P is described, as well as uploading the simulation to the tb.vcd file and entering the first two numbers. The figure shows only 2 input_a numbers, but 11 numbers will be entered (6 for the 1st module and 5 for the 2nd module).

Direct and inverse codes p_1, p_2 are defined:

$$p_1 = 231_{10} = 11100111_2 = E7_{16},$$
$$\overline{p_1} = 00011000_2 = 18_{16},$$
$$p_2 = 249_{10} = 11111001_2 = F9_{16},$$
$$\overline{p_2} = 00000110_2 = 06_{16}.$$

A simulation of the module's operation is given in Fig. 3, which shows the states of the inputs and outputs of the main elements of the accumulator at each clock.

To demonstrate the operation, a sequence of 11 numbers is defined as initial data, which can be divided into two streams. In this case, the following numbers, presented in hexadecimal form, are summed modulo p_1: 33, 67, 5E, 76, 70, 77 (odd positions in line input_a). Numbers 6A, 53, 5E, 2A, 2E are summed modulo p_2 (even positions in the line input_a).

The line tb.clk represents clock pulses. adder_8.out denotes the result of the sum $S'_{k,i} = A_i + S_{k,i-1}$ on the 8-bit adder; adder_9.out denotes the difference result $S''_{k,i} = (A_i + S_{k,i-1} - P_k)$ on the 9-bit adder; p_reverse denotes the inverse code of modulo p; control denotes the control input of the multiplexer; mux_out denotes the information outputs of the multiplexer.

By direct verification we are convinced that (33+67+5E+76+70) mod E7 = 1DE mod E7 = 10, and (6A+53+5E+2A+2E) mod F9 = 173 mod F9 = 7A, which confirms the correctness of the implementation of the software module. We can observe these values in the MUX 5 column, at 10 and 11 clock cycles.

```
`timescale 1ns/1ns  // Set the scale of the timeline
`include "ModSumTb.v"
module tb;
  reg clk;                  // Create a clock signal
  reg [7:0] input_a;        // Create a register for input_a
  reg [7:0] P_reverse;      // reversible module code P
  wire [7:0] mux_out;       // Create a wire for mux_out

  // Create an instance of the modulo_adder module
  modulo_adder modulo_adder(
    .input_a(input_a),
    .P(P_reverse),
    .clk(clk),
    .mux_out(mux_out)
  );
  always #5 clk = ~clk;     // Change the clock signal every 5 time units
  always @(posedge clk) begin          // switching the input P between P1 and P2
    if (P_reverse == 8'b00000110) begin
      P_reverse <= 8'b00011000;
      P <= 8'b11100111;
    end else if (P_reverse == 8'b00011000) begin
      P_reverse <= 8'b00000110;
      P <= 8'b11111001;
    end
  end
  initial begin
    P_reverse = 8'b00000110;   // initial P = P2
    $dumpfile("tb.vcd");
    $dumpvars(0, tb);
    clk = 0;                 // Initialize the clock signal

    //P1 = 11100111 (231)
    //P2 = 11111001 (249)
    //P1reverse = 00011000
    //P2reverse = 00000110
    #5;
    input_a = 8'b00110011;   // for 1 module
    #10;
    input_a = 8'b01101010;   // for 2 module

    // the following input_a is substituted here

  end
endmodule
```

Fig. 2 – Testing module

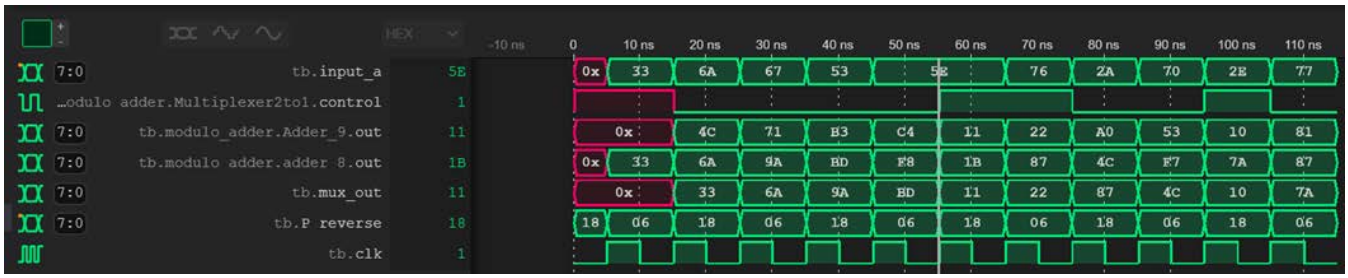

Fig. 3 – Simulation of the operation of a two-channel modulo adder

VI. Hardware implementation of the algorithm of accumulating

The hardware implementation of the proposed method of accumulating on logical modules of the type [20] and [21] is presented in Fig. 4.

The accumulator to arbitrary modulo contains 2n-bit and (n+1)-bit registers 1 and 3, respectively, where n is the bit width of the numbers being processed, n-bit and (n+1)-bit adders 2 and 4, multiplexer 5, first 6 and second 8 information inputs of the device, information outputs 9 of the device and clock input 7 of the device.

In the initial state, 2n-bit and (n+1)-bit registers 1 and 3 are zeroed.

Clock input 7 of the device receives clock pulses i=1, 2, 3, …, . Numbers A_i are supplied to the first information inputs 6 of the device with each clock pulse. The inverse code of module p_1 is supplied to the second information inputs 8 of the device on even clock cycles, and the inverse code of module p_2 is supplied on odd clock cycles (starting from 3). The sum $S_{1,i-1}$ modulo p_1 for numbers A_i (i=2, 4, 6,…) and the sum $S_{2,i-1}$ modulo p_2 for numbers A_i (i=1, 3, 5, …) is taken from the information 9 device outputs. In this case

$$0 \le A_i \ (i=2, 4, 6, \ldots) < p_1,$$
$$0 \le A_i \ (i=1, 3, 5, \ldots) < p_2.$$

On the first clock cycle of the device, the first *n*-bit number A_1 is written to the 2*n*-bit register 1. Moreover, it is written in the lowest *n* bits, and the highest *n* bits are reset to zero. From the lower n bits of the information outputs of the 2n-bit register 1, the number A_1 is supplied to the first information inputs of the n-bit adder 2, and the second information inputs receive a zero value from the highest n bits of the information outputs of the 2n-bit register 1. At the information outputs of *n*-bit adder 2, the value of the sum $A_1 + S_{1,0}$ is formed. Since the value of $S_{1,0}$ is 0 at the first clock cycle, the value A_1 is generated at the information outputs of *n*-bit adder 2. This value is then written to (*n*+1)-bit register 3 on the next clock cycle.

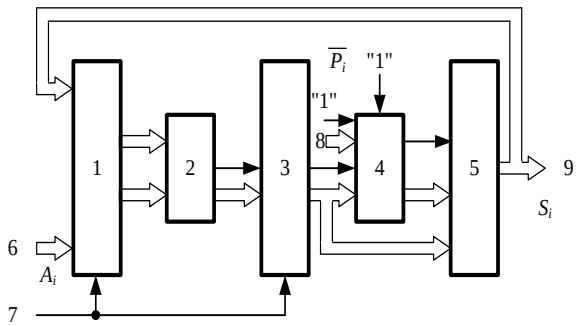

Fig. 4 – Hardware implementation of the accumulator to arbitrary modulo

On the second clock cycle of the device, the second *n*-bit number A_2 is written to the 2*n*-bit register 1, and the value of A_1 from the outputs of the *n*-bit adder 2 is written to the (*n*+1) bits of the (*n*+1)-bit register 3. The number A_2 is written to the lower *n* bits of the 2*n*-bit register 1, while a zero value is written to the highest *n* bits. From the lower *n* bits of the information outputs of the 2*n*-bit register 1, the number A_2 is supplied to the first information inputs of the *n*-bit adder 2, and the second information inputs receive a zero value from the highest *n* bits of the information outputs of the 2*n*-bit register 1. At the information outputs of *n*-bit adder 2, the sum value ($A_2 + S_{2,0}$) is formed. Since in the second cycle the value of $S_{2,0}$ is equal to 0, the value A_2 is formed at the information outputs of the *n*-bit adder 2. In this case, the first information inputs of the (*n*+1)-bit adder 4 will receive the value A_1 from the information outputs of the (*n*+1)-bit register 3, and the low-order *n* bits of the second information inputs will receive the *n*-bit inverse code of the modulus $\overline{p}_1$, which is complemented to the (*n*+1)th bit by the value of a logical one arriving at the (*n*+1)th bit of the second information inputs of the (*n*+1)-bit adder 4. Since a logical one signal is received at the carry input of the (*n*+1)-bit adder 4, this adder essentially performs a subtraction operation $(A_1 + S_{1,0}) - p_1$.

On the following cycles, the device operates in a similar way.

If the value $(A_i + S_{k,i-1}) \geq p_k$, then a carry signal is generated at the transfer output of the (*n*+1)-bit adder 4, which goes to the control input of the multiplexer 5 and connects its second information input with its information output, while the value $(A_i + S_{k,i-1}) - p_k$ is received at the second information input from the information outputs of the (*n*+1)-bit adder 4. If the value $(A_i + S_{k,i-1}) < p_k$, then the transfer signal at the transfer output of the (*n*+1)-bit adder 4 is absent and its first information input will be connected to the information output of the multiplexer, to which from the information output of the (*n*+1)-bit register 3 the value $(A_i + S_{k,i-1})$ arrives. As a result, at the information output of the multiplexer 5, and consequently at the information output 9 of the device, the values $S_{k,i}$ will be generated separately for the odd stream of numbers $S_{1,i}$ and for the even stream of numbers $S_{2,i}$. An example of how the device works is shown in Table 1.

TABLE 1. STATES OF THE INPUTS AND OUTPUTS OF THE ADDER ELEMENTS AT EACH CLOCK CYCLE

Clock No.	A_i	P_k	RG1 in	RG1 out	Adder 2	RG 3 out	Adder 4	MUX 5
1	33	0	33,0	33,0	33	–	–	–
2	6A	E7	6A,33	6A,0	6A	33	0,4C	33
3	67	F9	67,6A	67,33	9A	6A	0,71	6A
4	53	E7	53,9A	53,6A	BD	9A	0.B3	9A
5	5E	F9	5E,BD	5E,9A	F8	BD	0,C4	BD
6	5E	E7	5E,11	5E,BD	1B	F8	1,11	11
7	76	F9	76,22	76,11	87	1B	1,22	22
8	2A	E7	2A,87	2A,22	4C	87	0,A0	87
9	70	F9	70,4C	70.87	F7	4C	0.53	4C
10	2E	E7	2E,10	2E,4C	7A	F7	1.10	10
11	77	F9	77,7A	77,10	87	7A	0,81	7A

VII. RESULTS AND DISCUSSION

To compare the performance of the hardware implementation of the proposed algorithm of accumulating with existing solutions [11], we will assume that the response times of the main elements of the circuit are approximately the same for a width of 8 bits. As the bit width of the device increases, the delay in the adders will generally increase.

Let us introduce coefficient *k*, which takes into account the increase in the adder delay in relation to, for example, its 8-bit version. For various implementations of adders, the dependence of their performance on the bit width is studied in detail in [22].

Taking into account the above, the accumulating time T_1 of a sequence of *M* numbers for the known solution [11] can be represented as

$$T_1 = M(2t_{sum}k + t_{RG} + t_{mux}), \quad (11)$$

where t_{sum} is the delay time of the adder;

t_{RG} is the register response delay time;

t_{mux} is the multiplexer response delay time;

k is the coefficient taking into account the increase in the adder's bit width.

The addition time, taking into account the introduction of one more register to the circuit and taking into account the fact that during one cycle the addition of two numbers of the input stream occurs simultaneously, we will write it as

$$T_2 = M(2t_{sum}k + t_{RG} + t_{mux}) + (2t_{sum}k + t_{RG} + t_{mux}) \quad (12)$$

where the second group of terms takes into account the latent period of operation of the device.

Let us assume that for a device capacity of 8 bits, the response delay times of all elements are approximately the same and equal to $t=t_{sum}=t_{RG}=t_{mux}$.

Then (11) and (12) can be represented respectively as

$$T_1 = 2Mt(k+1), \quad (13)$$

$$T_2 = t(\frac{M}{2}(2k+1)+(k+3)). \quad (14)$$

We define the gain in performance as

$$B = \frac{T_1}{T_2} = \frac{2Mt(k+1)}{t(\frac{M}{2}(2k+1)+(k+3))} = \frac{(2M(k+1))}{\frac{M}{2}(2k+1)+(k+3)} \quad (15)$$

Figure 5 shows the results of calculating the performance gain for the device bit width of 8, 16, 3, 64 and 128 bits and the amount of numbers in the stream from 1 to 64.

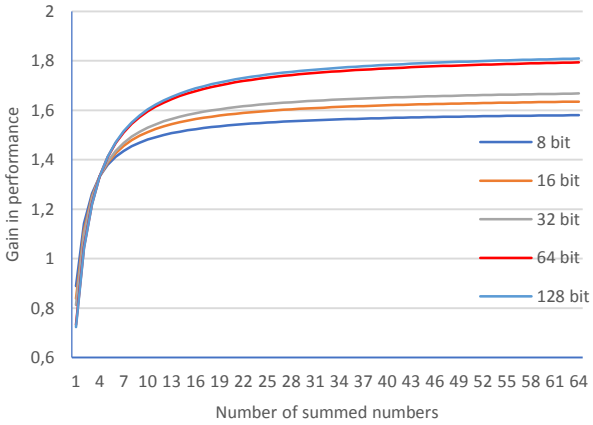

Fig. 5 – Gain in performance

Asymptotically, the gain tends to 2 as the amount of summed numbers increases.

VIII. Conclusion

The article presents a method of accumulating to arbitrary modulo for two streams of numbers from a common sequence of numbers.

For the proposed method of accumulating, an algorithm has been developed that implements it. The practical implementation of the algorithm is carried out using the Verilog HDL hardware description language. Using the developed testing module, the correct operation of the proposed algorithm was verified using specific numerical examples.

A hardware implementation of the proposed algorithm of accumulating to arbitrary modulo using binary adders, registers and a multiplexer is also proposed.

It is shown that the technical implementation has asymptotically 2 times better performance compared to existing solutions.

The use of the proposed method of accumulating in asymmetric cryptographic systems when implementing operations of multiplying large numbers modulo, all other things being equal, will allow the use of keys with a larger length, which will increase the systems' resistance to attacks.

References

[1] V. F. Shangin, *Zashchita informacii v komp'yuternyh sistemah i setyah* [Information protection in computer systems and networks]. DMK Press, 2012. 592 p. (in Russian).

[2] S. Singh, S. K. Maakar, and S. Kumar. "A Performance Analysis of DES and RSA Cryptography." *International Journal of Emerging Trends & Technology in Computer Science*, 2013, vol. 2, Issue 3, pp. 418–423.

[3] B. Preneel, "Cryptographic hash functions," *Eur. Trans. Telecommun.*, vol. 5, no. 4, pp. 431–448, 1994, doi: 10.1002/ETT.4460050406.

[4] G. J. Simmons, "A 'weak' privacy protocol using the rsa crypto algorithm," *Cryptologia*, vol. 7, no. 2, pp. 180–182, 1983, doi: 10.1080/0161-118391857900.

[5] A. Jung, "Implementing the RSA cryptosystem," *Comput. Secur.*, vol. 6, no. 4, pp. 342–350, Aug. 1987, doi: 10.1016/0167-4048(87)90070-8.

[6] J. Gordon, "Strong RSA keys," *Electron. Lett.*, vol. 20, no. 12, pp. 514–516, Jun. 1984, doi: 10.1049/EL:19840357.

[7] M. Preetha and M. Nithya, "A study and performance analysis of RSA algorithm," *IJCSMC*, Vol. 2, Issue. 6, June 2013, pg.126 – 139.

[8] Y. V. Artyukhov, *Analiz algoritma RSA. Nekotorye rasprostranyonnye elementarnye ataki i mery protivodejstviya im.* [Analysis of the RSA algorithm. Some common elementary attacks and countermeasures]. *Young Scientist*, no. 22, p. T.1. 85-87, 2010 (in Russian).

[9] H. Nikumbh and V. Shah, "Hardware implementation of modular multiplication," *2018 3rd IEEE Int. Conf. Recent Trends Electron. Inf. Commun. Technol. RTEICT 2018 - Proc.*, pp. 376–380, May 2018, doi: 10.1109/RTEICT42901.2018.9012447.

[10] B. V. Tarabrin, S.V. YAkubovski, N. A. Barkanov. *B. V. Tarabrin, Spravochnik po integral'nym mikroskhemam*. Ed. by B.V. Tarabrin. - 2nd ed., revised and enlarged - M.: Energia, 1981.

[11] V. I. Petrenko, J. V. Kuz'minov. *Nakaplivayushchij summator po modulyu* [Modulo Adder-Accumulator]. Patent Russia, no. 2500017 C1. 2013 (in Russian).

[12] V. I. Petrenko, D. D. Puiko. *Nakaplivayushchij summator po modulyu* [Modulo Accumulator]. Patent Russia, no. 2791441 C1. 2023 (in Russian).

[13] R. P. Brent and H. T. Kung, "A Regular Layout for Parallel Adders," *IEEE Trans. Comput.*, vol. C–31, no. 3, pp. 260–264, 1982, doi: 10.1109/TC.1982.1675982.

[14] T. Matsunaga and Y. Matsunaga, "Timing-constrained area minimization algorithm for parallel prefix adders," *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, vol. E90-A, no. 12, pp. 2770–2777, 2007, doi: 10.1093/IETFEC/E90-A.12.2770.

[15] P. Kogge and H. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Trans. Comput.*, vol. C–22, no. 8, pp. 786–793, 1973, doi: 10.1109/TC.1973.5009159.

[16] C. H. Pavan Kumar and K. Sivani, "Implementation of efficient parallel prefix adders for residue number system," *Int. J. Comput. Digit. Syst.*, vol. 4, no. 4, pp. 295–300, Oct. 2015, doi: 10.12785/IJCDS/040409.

[17] S. L. Harris and D. Harris, "Hardware Description Languages," *Digit. Des. Comput. Archit.*, pp. 170–235, 2022, doi: 10.1016/B978-0-12-820064-3.00004-0.

[18] "Circuit Modeling with Hardware Description Languages," *Top-Down Digit. VLSI Des.*, pp. 179–300, 2015, doi: 10.1016/B978-0-12-800730-3.00004-6.

[19] P. Benáček, V. Puš, H. Kubátová, and T. Čejka, "P4-To-VHDL: Automatic generation of high-speed input and output network blocks," *Microprocess. Microsyst.*, vol. 56, pp. 22–33, Feb. 2018, doi: 10.1016/J.MICPRO.2017.10.012.

[20] J. Zhu and N. Dutt, "Electronic System-Level Design and High-Level Synthesis," *Electron. Des. Autom.*, pp. 235–297, 2009, doi: 10.1016/B978-0-12-374364-0.50012-6.

[21] C. M. Maxfield, "'Traditional' Design Flows," *FPGAs: Instant Access*, pp. 75–106, 2008, doi: 10.1016/B978-0-7506-8974-8.00005-3.

[22] E.S. Balaka, D.A.Gorodecky, V.S. Rukhlov, A.N. Schelokov, *Razrabotka vysokoskorostnyh summatorov po modulyu na baze kombinacionnyh summatorov s parallel'nym perenosom* [Design and synthesis of high speed modulo adders using parallel prefix structure] Izvestiya SFedU. Engineering Sciences, no. 6 (179), p. 910, 2016 (in Russian).