Approaches for the performance increasing of software implementation of integer multiplication in prime fields

Vladislav Kovtun
Chair of Information Security
National Aviation University
Kiev, Ukraine
vladislav.kovtun@nrjetix.com

Andrew Okhrimenko
Chair of Information Security
National Aviation University
Kiev, Ukraine
andrew.okhrimenko@gmail.com

Authors have proposed the approach to increase performance of software implementation of finite field multiplication algorithm, for 32-bit and 64-bit platforms. The approach is based on delayed carry mechanism of significant bit in sum accumulating. This allows to avoid the requirement of taking into account the significant bit carry at the each iteration of the sum accumulation loop. The delayed carry mechanism reduces the total number of additions and gives the opportunity to apply the modern parallelization technologies.

Keywords: integer multiplication, software implementation, elliptic curve cryptosystem, cryptography, finite field, parallelism.

I. Introduction

The cryptographic transformations with public key has passed a long way since their introduction by Diffie and Hellman [1] to modern cryptosystems on algebraic curves, but the only things remains unchanged - operations in the number field $\mathbf{GF}(p)$. The integer multiplication takes the special place in number field operations, see fig. 1. Among the important problems of future development public key cryptosystems is the increasing of performance of software and hardware implementation. One of the approaches to the improvement cryptosystems performance is an increasing of performance of finite field arithmetics to be exact the multiplication.

Cryptographic transformations	Encryption/ decryption	Digital signature generation and verification	Key exchange		
Arithmetic in elliptic curve point group	Scalar multiplication of elliptic curve point				
	Point addition		Point doubling		
Arithmetic in finite field	Multiplication	Addition	Substruction	Squaring	Inversion
CPU commands	mov, mul, shr, shl, add, sub ...				

Fig. 1. Operation hierarchy of elliptic curve cryptosystem

It should be noted that the problem of the speed-up of arithmetic operation in number fields was deeply investigated by many scientists that is testified by significant number of publications in this area [2-8]. Except the arithmetic operations algorithms, are of interest the approaches to the architecture of the software libraries [9-18] with field operations, which allows decreasing significantly the overheads on fields operations in whole.

Publication analysis [2-7], allows to extract the most effective multiplication algorithms Comba [2, 3] and Karatsuba [3, 8, 10]. However, the Comba algorithm shows better results in performance tests (benchmark) of software implementations on modern platforms [3-9]. In article [8] describes the Karatsuba-Comba multiplication (KCM) algorithm for the RISC processors. The KCM algorithm is an interesting symbiosis of Comba and Karatsuba algorithms, where Karatsuba algorithm is used for the machine word multiplication only. As a result, the **main goal** of this article is a suggesting approaches to increase the effectiveness of software implementation of finite field $\mathbf{GF}(p)$ number multiplication (squaring) via a well-known Comba algorithm [2, 3, 8]. Among other things, such kind of investigations is evoked by the need to confirm the performance of software implementations of well-known algorithms with continuous development of modern 32-bit and 64-bit platforms. It is significant that during last ten years the direction of the multi-core processors and multiprocessor systems has been developed [8, 9].

II. MULTIPLICATION ALGORITHM-PROTOTYPE DESCRIPTION AND ITS MODIFICATION

The Comba algorithm [2] based on a main loops p. 2, p. 3 and nested loops p. 2.1, p.3.1. In the low level of hierarchy, in loops p. 2.1and p. 3.1 computes 64-bit integer product $(uv)^{(64)}$ which splits on two 32-bit integer $u^{(32)}$ and $v^{(32)}$.

The sum accumulation occurs in 32-bit temporary variables r_0, r_1 and r_2, on each iterations p. 2.1.2, p. 2.1.3.

The final result assignment and temporary variables r_0, r_1 and r_2 changing, occurs on each iteration on p. 2.2.

Algorithm. Comba's integer multiplication

```
Input: integers $a, b \in \mathbf{GF}(p)$, $w = 32$, $n = \log_{2^w} a$.
Output: $c = a \cdot b$
1. $r_0^{(32)} \leftarrow 0$, $r_1^{(32)} \leftarrow 0$, $r_2^{(32)} \leftarrow 0$.
2. For $k \leftarrow 0$, $k < 2n-1$, $k{+}{+}$ do
2.1. For $i \leftarrow 0$, $i < n$, $i{+}{+}$ do
2.1.1. For $j \leftarrow 0$, $j < n$, $j{+}{+}$
2.1.1.1. If $(i + j == k)$
2.1.1.1.1. $(uv)^{(64)} \leftarrow a_i^{(32)} \cdot b_j^{(32)}$.
2.1.1.1.2. $r_0^{(32)} \leftarrow r_0^{(32)} + v^{(32)}$, $r_1^{(32)} \leftarrow r_1^{(32)} + u^{(32)} + carry$, $carry \leftarrow 0$.
2.1.1.1.3. $r_2^{(32)} \leftarrow r_2^{(32)} + carry$, $carry \leftarrow 0$.
2.2. $c_k^{(32)} \leftarrow r_0^{(32)}$, $r_0^{(32)} \leftarrow r_1^{(32)}$, $r_1^{(32)} \leftarrow r_2^{(32)}$, $r_2^{(32)} \leftarrow 0$.
3. $c_{2n-1}^{(32)} \leftarrow r_0^{(32)}$.
4. Return $(c)$.
```

Consider the main drawbacks of Comba's algorithm:

- In nested loops p. 2.1 and p. 3.1 there is a sum accumulation with carry in 32-bit temporary variables r_0, r_1 and r_2, p. 2.1.2, p. 2.1.3 and p. 3.1.2, p. 3.1.3:
  2.1.2. $r_0^{(32)} \leftarrow r_0^{(32)} + v^{(32)}$, $r_1^{(32)} \leftarrow r_1^{(32)} + u^{(32)} + carry$, $carry \leftarrow 0$.
  2.1.3. $r_2^{(32)} \leftarrow r_2^{(32)} + carry$, $carry \leftarrow 0$.
  In this case there are 3 additions of 32-bit integer (includes 2 additions with carry), 3 assignments 32-bit variables r_0, r_1 и r_2. The sum accumulation with carry takes place in each iteration of loop p. 2.1.

- In nested loops p. 2.1 and p. 3.1, for the sum accumulation, for 32-bit variables r_0, r_1 and r_2 the transfers are considered, using the assembler code for the implementation of addition operation with carry. That in turn doesn't allow to pair and parallelize [11], as a result we observe an ineffective processor resource using.
- Loops p. 2 and p. 3 cannot be effectively parallelized due to high internal linkage code because of carry consideration.

Algorithm does not take into account a possibility of using modern processors support of 64-bit operations.

It is easy to obtain a computational complexity for the Comba's algorithm:

$$I_{mul}^{Comba} = 4I_{assign}^{32} + \left(\tfrac{n+1}{2}n + \tfrac{1+n-1}{2}(n-1)\right)$$
$$\left(1I_{mul}^{32} + 3I_{add}^{32} + 6I_{assign}^{32}\right) + 4(2n-1)I_{assign}^{32} =$$
$$= 4I_{assign}^{32} + n^2\left(1I_{mul}^{32} + 3I_{add}^{32} + 6I_{assign}^{32}\right)$$
$$+ 4(2n-1)I_{assign}^{32}$$

where I_{assign}^{32} - an assignment operation of 32-bit integers, I_{add}^{32} - an addition operation of 32-bit integers, I_{mul}^{32} - a multiplication operation of 32-bit integer.

Fig. 2 illustrates the drawbacks of algorithm for $n = 3$ and its impact on computational complexity of algorithm.

In upper part of figure there are two big numbers a and b represented by three 32-bit integers $a = (a_2, a_1, a_0)$ and $b = (b_2, b_1, b_0)$, where a_i and b_i have a machine word bit size. Algorithm iterations are presented under the solidus. It should be noted that algorithm Comba implements long multiplication technique, known from school, with small difference: the multiplier part a_i $i = \overline{1, n}$ multiply on all parts of other multiplier b_j $j = \overline{1, n}$, in case of fulfillment the condition $(i + j == k)$ (in columns).

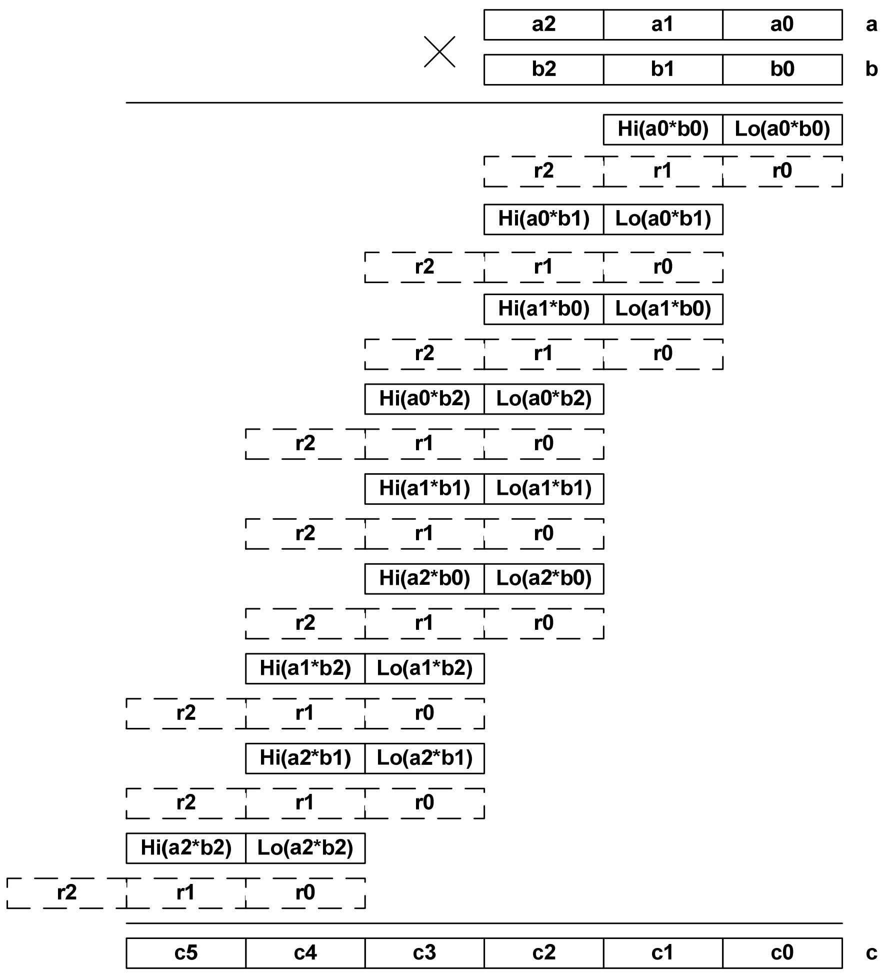

Fig. 2. Graphic interpretation of Comba algorithm

Such approach does not lead to strings addition (multiplication of intermediate results) as a long multiplication, but to columns addition. That allows to find a part of resulting product c_i (under the solidus). As shown in the fig. 2, each multiplication is accompanied by the sum accumulation with a carry.

The computational complexity for $n = 3$, will be:

$$I_{mul}^{Comba} = 4I_{assign}^{32} + 9\left(1I_{mul}^{32} + 3I_{add}^{32} + 6I_{assign}^{32}\right) + \; + 20I_{assign}^{32} = 78I_{assign}^{32} + 9I_{mul}^{32} + 27I_{add}^{32} .$$

Now let’s consider the approaches suggested by the authors addressed to eliminate the drawbacks:

- The modern 32-bit processors effectively implement the addition operations of 32-bit and 64-bit integers, using 64-bit or 32-bit commands. That allows to implement a carry accumulation by addition of 32-bit variables in 64-bit variable-accumulator, that save the carry accounting and correction requirements after the addition with variables r_0, r_1 and r_2. An accumulated carry will be accounted in the final iterations of the loops in p.2 and p.3.
- Modern processors have multi-core architecture; that allows them to execute several instruction flows at the same time. This property brings to parallel execute of iterations in loop p.2 and p.3 by the OpenMP library [11-13].

Following notations are to be introdused: through $t^{(64)}$ will symbolized 64-bit variables, and through $t^{(32)}$ - 32-bit variables; operation $\mathrm{hi}_{(32)}\left(t^{(64)}\right)$ extracts 32 the most significant bits in 64-bit variable, and operation $\mathrm{low}_{(32)}\left(t^{(64)}\right)$ extracts 32 the least significant bits in 64-bit variable.

Algorithm. Modified Comba's integer multiplication

```
Input: целое $a, b \in \mathbf{GF}(p)$, $w = 32$, $n = \log_{2^w} a$, $nk = 2n - 1$.
Output: $c = a \cdot b$
1. $r_0^{(64)} \leftarrow 0$, $r_1^{(64)} \leftarrow 0$, $r_2^{(64)} \leftarrow 0$.
2. For $k \leftarrow 0$, $k < n$, $k++$ do
2.1. For $i \leftarrow 0$, $j \leftarrow k$, $i \le k$, $i++$, $j--$ do
2.1.1. $(uv)^{(64)} \leftarrow a_i^{(32)} \cdot b_j^{32}$.
2.1.2. $r_0^{(64)} \leftarrow r_0^{(64)} + v^{(32)}$, $r_1^{(64)} \leftarrow r_1^{(64)} + u^{(32)}$.
2.2. $r_1^{(64)} \leftarrow r_1^{(64)} + \mathrm{hi}_{(32)}\left(r_0^{(64)}\right)$, $r_2^{(64)} \leftarrow r_2^{(64)} + \mathrm{hi}_{(32)}\left(r_1^{(64)}\right)$.
2.3. $c_k^{(32)} \leftarrow \mathrm{low}_{(32)}\left(r_0^{(32)}\right)$, $r_0^{(64)} \leftarrow \mathrm{low}_{(32)}\left(r_1^{(32)}\right)$, $r_1^{(64)} \leftarrow \mathrm{low}_{(32)}\left(r_2^{(32)}\right)$, $r_2^{(64)} \leftarrow 0$.
3. For $k \leftarrow n$, $l \leftarrow 1, k < nk$, $k++$, $l++$ do
3.1. For $i \leftarrow l$, $j \leftarrow k - l$, $i < n$, $i++$, $j--$ do
3.1.1. $(uv)^{(64)} \leftarrow a_i^{(32)} \cdot b_j^{32}$.
3.1.2. $r_0^{(64)} \leftarrow r_0^{(64)} + v^{(32)}$, $r_1^{(64)} \leftarrow r_1^{(64)} + u^{(32)}$.
3.2. $r_1^{(64)} \leftarrow r_1^{(64)} + \mathrm{hi}_{(32)}\left(r_0^{(64)}\right)$, $r_2^{(64)} \leftarrow r_2^{(64)} + \mathrm{hi}_{(32)}\left(r_1^{(64)}\right)$.
3.3. $c_k^{(32)} \leftarrow \mathrm{low}_{(32)}\left(r_0^{(32)}\right)$, $r_0^{(64)} \leftarrow \mathrm{low}_{(32)}\left(r_1^{(32)}\right)$, $r_1^{(64)} \leftarrow \mathrm{low}_{(32)}\left(r_2^{(32)}\right)$, $r_2^{(64)} \leftarrow 0$.
4. $c_{nk}^{(32)} \leftarrow \mathrm{low}_{(32)}\left(r_0^{(32)}\right)$.
5. Return $(c)$.
```

It is not difficult to get a computational complexity of modified Comba algorithm:

$$\begin{aligned} I_{mul}^{Mod.\,Comba} &= 4I_{assign}^{64} + \left(\tfrac{n+1}{2}n + \tfrac{1+n-1}{2}(n-1)\right) \\ &\left(1I_{mul}^{32} + 2I_{add}^{64|32} + 2I_{assign}^{64}\right) \\ &+ (2n-1)\left(2I_{add}^{64|32} + 1I_{assign}^{64} + 1I_{assign}^{32}\right) = \\ &= 4I_{assign}^{64} + n^2\left(1I_{mul}^{32} + 2I_{add}^{64|32} + 2I_{assign}^{64}\right) \\ &+ (2n-1)\left(2I_{add}^{64|32} + 1I_{assign}^{64} + 1I_{assign}^{32}\right) \end{aligned},$$

where I_{assign}^{32} - an assignment operation of 32-bit integers, I_{assign}^{64} - an assignment operations of 64-bit integers, I_{add}^{32} - an addition operation of 32-bit integers, $I_{add}^{64|32}$ - an addition operation of 32-bit and 64-bit integers, I_{mul}^{32} - a multiplication of 32-bit integers.

Fig. 3, 4 illustrate the algorithm 2 for $n = 3$; computational complexity for this case will be: $I_{mul}^{Mod.\,Comba} = 27I_{assign}^{64} + 9I_{mul}^{32} + 28I_{add}^{64|32} + 5I_{assign}^{32}$.

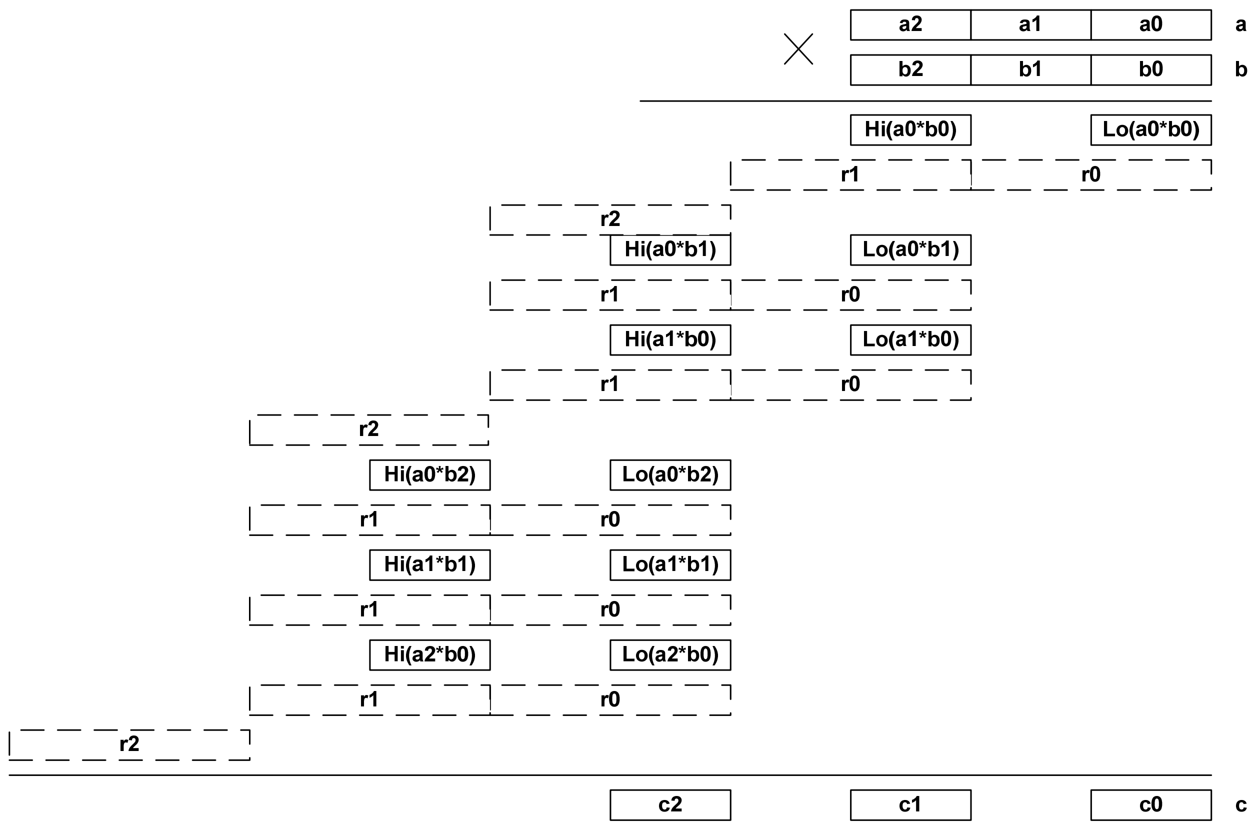

Fig. 3. Graphic interpretation of loop 2 in Modified Comba algorithm

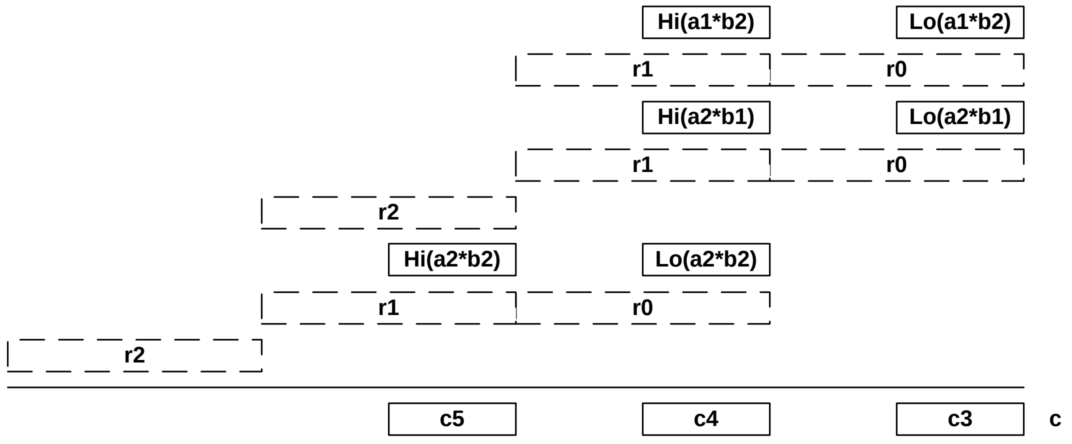

Fig. 4. Graphic interpretation of loop 3 in Modified Comba algorithm

Comparison with the other algorithms. For the relevant comparison of attained results, the authors reviewed well-known software math libraries [14-24] for public key cryptography. According to the review results the software library GMP was chosen as an etalon [14]. It should be noted that GMP uses Karatsuba multiplication algorithm for the integer multiplication [2]. The comparison of software implementations will be carrying out by comparing average time execution of software implementation of Comba, modified Comba algorithms and implemented in GMP library for one million iterations.

The performance measurement of algorithm software implementation is proposed to be conducted for the fields from [25], except $\mathbf{GF}(p82)$ field. These fields recommended for the usage in cryptographic application for the different security levels provisioning. In table 1 we will indicate the brief definition of fields and prime modules.

Fields for which perfmormance measurements are made Table 1

Field	Prime modulo
GF(p82)	500000000000000008503491
GF(p164)	24999999999994130438600999402209463966197516075699
GF(p192)	6277101735386680763835789423 176059013767194773182842284081
GF(p224)	26959946667150639794667015087019630673557916260026308143510066298881
GF(p256)	115792089210356248762697446949407573530086143415290314195533631308867097853951
GF(p320)	4271974071841820164790042159200669057836414062331724137933565193825968686576267080087081984838097
GF(p384)	3940200619639447921227904010014361380507973927046544666794690527962765939911326356939895 6308152294913554433653942643
GF(p521)	6864797660130609714981900799081393217269435300143305409394463459185543183397655394245057746333217197532963996371363321113864768612440380340372808892707005449

The proposed modified algorithm Comba and its prototype – algorithm Comba have been implemented in C++, compiled with Microsoft Visual Studio 2010 in Release Win32 configuration with Maximize Speed parameter and SSE2 instruction support.

The etalon library GMP v4.1.2 compiled with Microsoft Visual Studio .NET but instrumental application compiled with Microsoft Visual Studio 2010 in Release Win32 configuration with Maximize Speed parameter and SSE2 instruction support.

Tested by mainstream mobile platform with Intel Core i3 350M CPU and desktop platform with Intel Pentium Dual Core E5400.

Performance measurement timings for the different algorithms, implementations and CPU are shown in Table 2.

Running time of multiplication software implementation without modulo reduction Table 2

Field	**Time, µs**					
	Core i3			**Pentium Dual Core**		
	Mod. Comba	**Comba**	**GMP4.1**	**Mod. Comba**	**Comba**	**GMP4.1**
GF(p82)	0,075	0,120	0,121	0,0687	0,119	0,125
GF(p164)	0,21	0,393	0,4	0,209	0,363	0,407
GF(p192)	0,276	0,393	0,41	0,289	0,363	0,414
GF(p224)	0,343	0,69	0,549	0,364	0,59	0,522
GF(p256)	0,422	0,875	0,638	0,456	0,744	0,648
GF(p320)	0,6973	1,278	0,97	0,686	1,053	0,969
GF(p384)	0,961	1,75	1,38	0,94	1,45	1,36
GF(p521)	1,63	2,8	2,663	1,486	2,41	2,643

As can be seen from the timing in Table 2, the proposed modification of the algorithm Comba allowed to reach the advantage of 1.5 times above the GMP. Classical implementation of algorithm Comba appeared to be the slowest, that is confirmed by the theoretical estimation (contains a larger number of addition and assignment operations). In addition, proposed software implementations of multiplication algorithms arreared to be more efficient on Dual Pentium CPU with higher frequency then on Core i3 CPU with several instruction streams. These implementations of multiplication algorithms do not support parallelization, thus a more powerful multicore CPU Core i3 with 4 instructions processing flows was not able to realize its full potential.

Conclusions. Following the results of the research next conclusions can be drawn:

1. Proposed approach of delayed carry, allows to increase the performance of software implementation of Comba integer multiplication algorithm by 1.5-2 times and surpass the performance of the popular math library GMP v4.1.2, average by 1.5 times.

2. Modified multiplication Comba algorithm is more preferred than Karatsuba algorithm [2] which used in GMP library, because implementation of modified Comba algorithm is faster than Karastuba [2] implementation in GMP for the modern hardware platform (32 & 64-bit).

3. Delayed carry mechanism allows to apply different parallelization techniques to the modified Comba algorithm, for example OpenMP [28], Intel Threading Blocks [30], OpenCL [29].

Recently, the microprocessors development is directed at increasing the number of instruction processing flows. Thus suitable algorithms should be developed for perspective microprocessors should develop for efficient parallelization implementation by perspective micro.

nVidia company, proposes GPU with more than 256 cores and suitable CUDA Toolkit [27] which allows to implement valid multithread applications. A great part of attention is pied already to this direct and this article is another illustration [9]. A further research course will focus on investigation and effective parallelization of algorithms for arithmetic operations with integers.

References

[1] Diffie W., Hellman M. E., "New directions in cryptography," IEEE Transactions on Information Theory, vol. IT-22, pp. 644–654, 1976.

[2] Comba P. G. Exponentiation cryptosystems on the IBM PC // IBM Systems Journal. –Vol. 29(4). -1990. -pp. 526–538.

[3] Brown M., Hankerson D., Lopez J., Menezes A. Software implementation of the NIST elliptic curves over prime fields // Research Report CORR 2000–55. Department of Combinatorics and Optimization, University of Waterloo. –Canada: Waterloo, Ontario, 2000. –21p.

[4] Hong S-M., Oh S-Y., Yoon H. New Modular Multiplication algorithms for fast modular exponeniation // Advances in Cryptology-Proceedings of Eurocrypt '96. –Springer-Verlag. -1996. –pp.166-177.

[5] Avanzi R. M. Aspects of hyperelliptic curves over large prime fields in software implementations // Cryptology ePrint Archive. –Report 2003/253. –2003. –23p. Available at: http://eprint.iacr.org

[6] Paar C. Implementation options for finite filed arithmetic for elliptic curve cryptosystems // Worchester Polytechnic Institute. –ECC'99. –1999. –31p. Available at: http://www.ece.wpi.edu/research/crypto.html

[7] Gaubatz G. Versatile Montgomery multiplier architectures. Master thesis: electrical and computer engineering. –2002. –Worcester polytechnic institute. –101p.

[8] Johann Großschadl, Roberto M. Avanzi, Erkay Sava, Stefan Tillich. Energy-Efficient Software Implementation of Long Integer Modular Arithmetic // Advances in Cryptology-Prociding in CHES'2005. –Springer-Verlag. -2005. -LNCS 3659. -pp.75-90.

[9] Giorgi P. Izard T, Tisserand A. Comparison of Modular Arithmetic Algorithms on GPUs. URL: http://hal-lirmm.ccsd.cnrs.fr/lirmm-00424288/fr/

[10] Weimerskirch A., Paar C. Generalizations of the Karatsuba Algorithm for Efficient Implementations. // Cryptology ePrint Archive. –Report 2006/224. –2006. –17p. Available at: http://eprint.iacr.org

[11] Intel® 64 and IA-32 Architectures Optimization Reference Manual. Order Number: 248966-025. Available at: http://intel.com

[12] The OpenMP API Specification for Parallel Programming. Available at: http://openmp.org

[13] OpenMP in Visual C++. Available at: http:// http://msdn.microsoft.com/en-us/library/tt15eb9t.aspx

[14] The GNU Multiply Precision Library (GMP). URL: http://gmplib.org

[15] LiDIA. URL: https://www.cdc.informatik.tu-darmstadt.de/en/cdc

[16] Multiprecision Unsigned Number Template Library (MUNTL). URL: http://mktmk.narod.ru/eng/muntl/muntl.htm

[17] TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. URL: http://discovery.csc.ncsu.edu/software/TinyECC

[18] Galois Field Arithmetic Library. URL: http://www.partow.net/projects/galois/

[19] MPFQ: Fast Finite Fields Library. URL: http://mpfq.gforge.inria.fr/

[20] BBNUM. URL: http://www.iw-net.org/index.php?title=Bbnum_library

[21] FLINT: Fast Library for Number Theory. URL: http://www.flintlib.org

[22] Multiprecision Integer and Rational Arithmetic C/C++ Library (MIRACL). URL: http://indigo.ie/~mscott

[23] LibTom Projects: LibTomMath, TomsFastMath. URL: http://libtom.org

[24] Abusharekh A., Gaj K. Comparative Analysis of Software Libraries for Public Key Cryptography // Software Performance Enhancement for Encryption and Decryption, SPEED'2007. June 11-12, 2007.

[25] Giorgi P., Imbert L., Izard T. Multipartite Modular Multiplication. Preprint. URL: http://hal.archives-ouvertes.fr/lirmm-00618437/fr/

[26] National Institute of Standards and Technology, Recommended Elliptic Curves for Federal Government Use, Appendix to FIPS 186-2, 2000. –43p.

[27] NVIDIA. NVIDIA CUDA Programming Guide 2.0. 2008.

[28] OpenMP. The OpenMP® API specification for parallel programming. Available at: http://openmp.org

[29] OpenCL - The open standard for parallel programming of heterogeneous systems. Available at: http://www.khronos.org/opencl

[30] Intel Threading Blocks. Available at: http://software.intel.com/en-us/articles/intel-tbb