Scalable Normal Basis Arithmetic Unit for Elliptic Curve Cryptography

J. Schmidt, M. Novotný

The design of a scalable arithmetic unit for operations over elements of $GF(2^m)$ represented in normal basis is presented. The unit is applicable in public-key cryptography. It comprises a pipelined Massey-Omura multiplier and a shifter. We equipped the multiplier with additional data paths to enable easy implementation of both multiplication and inversion in a single arithmetic unit. We discuss optimum design of the shifter with respect to the inversion algorithm and multiplier performance. The functionality of the multiplier/inverter has been tested by simulation and implemented in Xilinx Virtex FPGA. We present implementation data for various digit widths which exhibit a time minimum for digit width $D = 15$.

Keywords: finite fields, normal base, multiplication, inversion, arithmetic unit.

1 Introduction

Contemporary cryptographic schemata are frequently based on the *Discrete Logarithm Problem* (DLP): find integer k such that

$$Q = k \cdot P = \sum_{1}^{k} P \qquad (1)$$

for given group elements P and Q.

In elliptic curve cryptography (ECC), P and Q are points on a chosen elliptic curve over a finite field. We focus on curves over $GF(2^m)$, where point coordinates are expressed as m-bit vectors.

The DLP in such a group is exponentially hard in comparison with DLP in a multiplicative group over a finite field. This means that a 173 bit key provides approximately the same security level as the 1024-bit RSA [7]. This fact is very important in applications such as chip cards, where the size of the hardware and energy consumption is crucial.

In algorithms such as the Elliptic Curve Digital Signature Algorithm (ECDSA), k is an m-bit integer, P is a chosen point and Q is computed using Eq. 1. It requires addition, multiplication and inversion over $GF(2^m)$.

1.1 Finite field operations

The implementation of these operations is determined by the representation of the field elements in $GF(2^m)$ [2]. In this work we focus on the normal basis representation.

Addition over elements of $GF(2^m)$ is implemented as a bit-wise XOR. *Squaring* is realized by rotation (cyclic shift) one bit to the right. Because it is so simple (one clock cycle), it is regarded as a special case. *Multiplication* is based on matrix multiplication over $GF(2^m)$. In hardware, a special unit (multiplier) is necessary. The best-known algorithm for *inversion* in normal basis is the algorithm of Itoh, Teechai and Tsujii (ITT) [3] based on multiplication and squaring.

1.2 Scalability

We understand *scalability* as the ability to change scale or to be available in various versions. The term 'scale' can be paraphrased as 'the important dimension' and hence is a matter of viewpoint.

The basic dimension of cryptography is the measure of security, which translates to key length. A unit is scalable if it can serve for computations with varying key length, provided that the internal memory is sufficient [5]. We suggest calling this kind of scalability *scalability in precision*.

In the world of parallel processing and VLSI design, the important dimension is the number of processors or other hardware units. An algorithm scales well if we can obtain more performance by assigning more resources (processors, chip area). We shall speak about *scalability in performance*. As this paper is focused on this kind of scalability only, we will use just the term 'scalability'.

We need scalability for two reasons. The first (and more important) reason is practical: hardware implementations are needed for a variety of contexts, from smart-card devices to high-throughput servers. The second reason is a fair comparison of design versions, where scaling the units to match in one dimension is preferable to artificial quality factors.

To scale a unit means to employ parallelism at a certain level of abstraction. We can scale at algorithmic level by selecting a different algorithm, at the register-transfer level, e.g., by changing the data path width, or at the gate level by factoring combinational circuits, or even at the physical level. This work aims at the seam between algorithmic and register-transfer level.

Units composed of sub-units are harder to scale. We might be lucky and find a unified scaling parameter, e.g., data path width. In the general case, however, each sub-unit may have a different scaling parameter.

The sub-units interact. Firstly, if there is a common clock (as preferred in practice), it must suit the slowest sub-unit. Secondly, to achieve the best global area-performance ratio, local area-performance tradeoffs are not independent.

The above sketched elliptic curve operations offer very limited parallelism and therefore cannot be a major source of scalability. Scalability must be sought for in finite field operations, and therefore the most interesting area for scalability in an elliptic curve processor is the finite field unit.

1.3 Metrics

The metrics used reflects the abstraction level of the parallelism employed. When the data path units are simple and their implementations are obvious, we can get at a lower

level. In this work, we use the classical metrics of area (which is connected to power consumption) and time.

When a unit is scaled, its area A and time t vary in opposite directions. Time t depends on the number of clock cycles T spent in a given calculation and on the critical path length τ in hardware. To compare differently scaled units, we use the quality measure

$$Q = At = AT\tau.$$

2 Previous work

Massey and Omura proposed a multiplier [6] that employs the regularity of equations for all bits of a result. From the equation for one bit of a result (e.g. c_0), equations for other bits can be derived by rotating bits of the arguments a and b [2]. In this multiplier, one bit of the result is computed completely in one clock cycle. Then, registers holding the arguments a and b are rotated right one bit between cycles. The computation of m bits of the result takes m clock cycles and hence this multiplier is also called bit-serial.

Agnew, Mullin, Onyszchuk and Vanstone introduced a modification of the Massey-Omura multiplier [1] (in this paper, we call it the AMOV multiplier). They divided the equation for each bit c_i into m products $P_{i,j}$:

$$c_i = P_{i,0} + P_{i,1} + \ldots + P_{i,m-2} + P_{i,m-1} \qquad (2)$$

In the first clock cycle, the product $P_{i,i+0}$ of bit c_i for all $i \in \langle 0, m-1 \rangle$ is evaluated. In the next cycle, the product $P_{i,i+1}$ (all subscripts are reduced mod m) of bit c_i for all $i \in \langle 0, m-1 \rangle$ is evaluated and added to the intermediate result, and so on. All bits of the result are successively evaluated in parallel; the computation is pipelined.

The block diagram of the AMOV multiplier is in Fig. 1. The multiplication is performed as follows: In the first step, both operands a and b are loaded from inputs IN_1 and IN_2 to registers A and B, respectively.

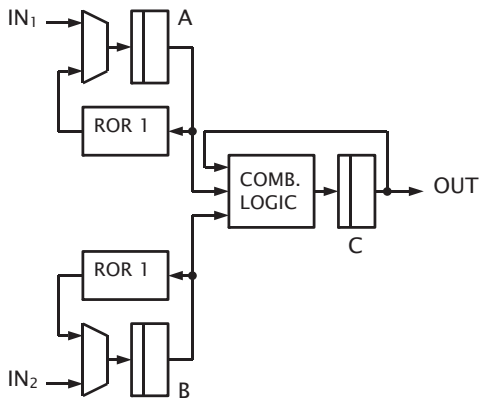

Fig. 1: AMOV multiplier

Then, in each of the following m clock cycles, both the A and B registers are rotated right (this is represented by the blocks *ROR 1*) and the result c in register C is evaluated successively by the block *COMB. LOGIC*, which implements the products $P_{i,j}$ from Eq. 2. After m clock cycles, the result $c = a \times b$ is available at output *OUT*. All registers and data paths are m bits wide.

The amount of hardware is the same as for the non-pipelined Massey-Omura multiplier, but the critical path is short and constant (it does not depend on m) and so the maximum achievable frequency is higher. This multiplier is widely used.

The computation of an inverse element (inversion) by the ITT algorithm [3] is usually controlled by a microprogram [4]. When implementing the ITT inversion using a classical AMOV multiplier, additional registers and data transfers outside the multiplier are necessary.

In this work we present a modification of the AMOV multiplier, which allows efficient implementation of both the multiplication and ITT inversion algorithms. In comparison with the microprogrammed inversion, no additional registers or data transfers outside the multiplier are necessary. We also introduce several improvements to this multiplication/inversion unit, which lead to increased performance and a better performance/area ratio.

3 Structure of the unit

The data path of our arithmetic unit is an extension of the AMOV multiplier. By adding one more input to the multiplexer preceding register A and by redirecting some data paths (see bold lines in Fig. 2), we can simply implement both multiplication and ITT inversion in the unit and thus save additional registers and data transfers outside of the multiplier.

The modified AMOV multiplier has a dedicated control unit based on a finite state machine, two counters *COUNT_INV* and *COUNT_K* and the shift register M. It implements the commands *load_op*, *multiply* and *invert*.

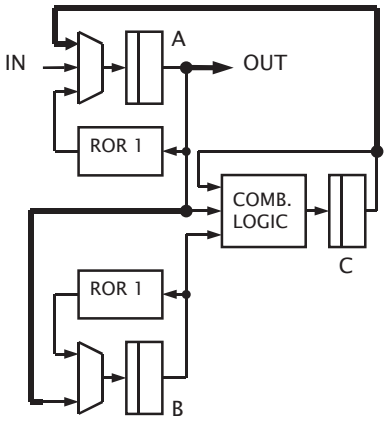

Fig. 2: Modified AMOV multiplier

3.1 Multiplication

Multiplication is performed as follows: In the first two steps, both operands a and b are successively loaded from input IN to registers A and B. Then, in m clock cycles, the multiplication is performed as in the standard AMOV multiplier. After its completion, the result $c = a \times b$ is loaded from register C to register A and is available at output *OUT*.

Algorithm 1: An implementation of the ITT inversion in the modified AMOV multiplier

1. $M_r...M_0 \leftarrow m-1$;
2. $A \leftarrow IN$;
3. for *COUNT_INV* in $r-1$ down to 0 do
   - 3.1 $B \leftarrow A$;
   - 3.2 for *COUNT_K* in $M_{2r-1} \ldots M_r$ down to 1 do
     - 3.2.1. $B \leftarrow B$ ror 1;
   - 3.3 C $\leftarrow B \times A$;
   - 3.3a $A \leftarrow C$;
   - 3.3b $M \leftarrow M$ shl 1;
   - 3.4 if M_r = ‚1' then
     - 3.4.1 $B \leftarrow A$;
     - 3.4.2 $B \leftarrow B$ ror 1; $A \leftarrow IN$;
     - 3.4.3 $C \leftarrow B \times A$;
     - 3.4.4 $A \leftarrow C$;
4. $A \leftarrow A$ ror 1;

3.2 Inversion

Our unit computes the ITT inversion [2], [3] by Algorithm 1. It comprises $\lfloor \log_2(m-1) \rfloor + w(m-1) - 1$ multiplications, where $w(\mathrm{m}-1)$ is the number of non-zero bits in the binary representation of $m-1$. Furthermore, it needs $(m-1) - w(m-1)$ squarings. The total number of clock cycles spent for one inversion is

$$C_{\mathrm{INV}} = \left(\lfloor \log_2(m-1) \rfloor + w(m-1) - 1\right) \cdot C_{\mathrm{MUL}} + \\ + \left((m-1) - w(m-1)\right) \cdot C_{\mathrm{SQR}} + const,$$

where C_{MUL} is number of clock cycles of 1 multiplication ($C_{\mathrm{MUL}} = m$) and C_{SQR} is number of clock cycles of one squaring ($C_{\mathrm{SQR}} = 1$).

Note that the inverted value must be available at input *IN* during the whole process of inversion. The rotation capability of register *B* is used for the computation of squarings in steps 3.2.1 and 3.4.2.

4 Scaling the multiplication

The basic ECC operation (Eq. 1) is performed by successive point additions and point doublings. Each of these operations needs 1 inversion, 2 multiplications and 1 squaring [2]. The number of clock cycles necessary for one point addition or doubling is then:

$$C_{\mathrm{PADD}} = \left(\lfloor \log_2(m-1) \rfloor + w(m-1) + 1\right) \cdot C_{\mathrm{MUL}} + \\ + \left(m - w(m-1)\right) \cdot C_{\mathrm{SQR}} + const. \qquad (3)$$

We can reduce the number of clock cycles C_{PADD} in two ways: by reducing the number of clock cycles C_{MUL} of the multiplications and the number of clock cycles C_{SQR} of the iterative squaring.0

Both the Massey-Omura and the AMOV multipliers need m clock cycles for computing all m bits of the result. Some authors also call them bit-serial, because they compute one bit of a result in one clock cycle.

There is a digit-serial variant of the Massey-Omura multiplier (some authors call it sliced or parallel). In this multiplier, D bits (also called a digit) are evaluated in one clock cycle. In the case of a digit-serial AMOV multiplier, D products $P_{i,j}$ are evaluated in one clock cycle. All m bits of the result are then evaluated in $C_{\mathrm{MUL}} = \lceil m/D \rceil$ cycles.

Since more products are evaluated in one clock cycle, more combinational logic is necessary. The size of the block *COMB. LOGIC* in Fig. 2 is proportional to $D+1$. The size of other blocks remains constant.

As the combinational logic becomes more complex, the length of the critical path grows proportionally to $\log D$. Since one multiplication needs m/D clock cycles, the total time necessary for one multiplication is $O\left((m/D) \times \log D\right)$, and the total time of one inversion (or point addition on elliptic curve) is

$$T_{\mathrm{PADD}} = O\left((\log m \cdot \frac{m}{D} + m).\log D \right). \qquad (4)$$

5 Scaling the iterative squarings

Another way to improve the performance of the ITT inversion (and consequently the point addition) is to reduce the number of clock cycles necessary for the iterative squarings in step 3.2.1 of Algorithm 1.

Adding one or more blocks performing "long distance" rotations can reduce the number of clock cycles required for all iterative squarings (Fig. 3). We shall refer to the hardware realizing these rotations as the *shifter.*

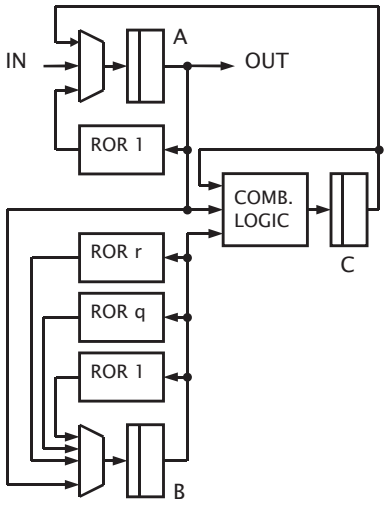

Fig. 3: "Long distance" rotations form a shifter that saves clock cycles necessary for squarings in ITT inversion

Let m be the degree of the finite field we work in, $GF(2^m)$. Let

$$m-1 = (b_r b_{r-1} \ldots b_1 b_0)$$

be the binary representation of m-1 such that the most significant bit b_r=1. The rotations required by the Itoh-Teechaji-Tsujii algorithm are

$$K = \{k_i, i = 1 \ldots r\}, k_i = (b_r \ldots b_i).$$

The binary representation of k_i is $b_r \ldots b_i$. Each of the shifts is performed exactly once in an inversion operation.

Let A, T, and τ be the area, the total number of clock cycles spent in rotations, and the critical path of the shifter. Let A_0, T_0, and τ_0 be the area, total number of clock cycles spent in

multiplications, and the critical path of the rest of the arithmetic unit. The quality measure of the entire unit, and hence our optimization criterion, is

$$Q = (A + A_0)(T + T_0) \cdot \max\{\tau, \tau_0\} \quad (5)$$

This equation also shows the two dependencies between the shifter and the multiplier. Firstly, the ratio of the shifter's area and time must be 'the right one'. For each A_0, T_0 and τ_0, the area and time of the shifter shall be adjusted to achieve minimum Q of the entire unit.

Secondly, the shifter may slow down the AU clock. In this case, not only the time $T\tau$ is longer, but also the multiplication time is $T_0\tau$ instead of $T_0\tau_0$. As the multiplier dominates, the penalty may become unacceptable.

The multiplier is scaled by manipulating the digit size, which affects A_0, T_0 and τ_0, while changing the number of rotations in the shifter varies A, T, and τ. Thus the optimization problem becomes multi-parametric. Because the multiplier logic dominates in both area and time, we solve it in a Pareto-optimal way: we modify the shifter to achieve optimum total Q for a given multiplier digit width.

5.1 Decomposition in time and space

To implement a set of rotations, one might use:

- a multiplexer structure, such as the barrel shifter, spending a single clock cycle for each rotation, or
- hardware providing the rotation by 1 only, requiring k clock cycles to rotate by k.

These options can be seen as decompositions in space and in time, respectively. In a more general approach, we construct hardware providing a limited set of rotations, and we use it in multiple clock cycles to realize the given rotations.

The solution of our problem can be decomposed as follows:

- Find rotations $s_j \in \mathbf{Z}^+$, $j = 1 \ldots n$ and factors $T_{ij} \in \mathbf{Z}^+$, $i = 1 \ldots r$, $j = 1 \ldots n$ such that

$$\sum_{j=1}^{n} s_j T_{ij} \equiv k_i \bmod m \quad (6)$$

(the *time domain problem*).

- Implement the rotations s_j under the optimality criterion in Eq. 5 (the *space domain problem*).

For such a sequential decomposition to work, the first step must estimate the quality of the result of the second step. In our case, we need the estimation of area, clock cycles, and critical path of a circuit realizing a given set of rotations. Thanks to the special nature of rotations k_i, we can have made reasonable assumptions about them.

The decomposition of the problem is summarized in Fig. 4.

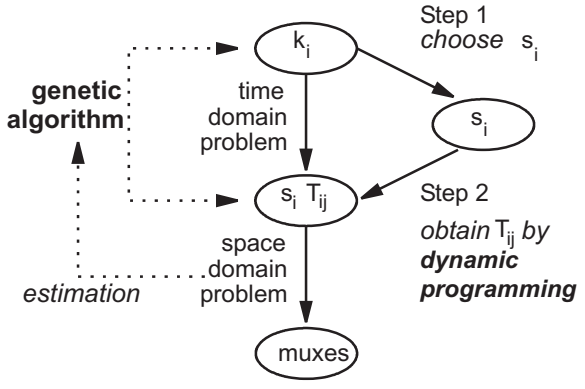

Fig. 4: Decomposition

5.2 Space domain problem

Let N_{MUX} be the number of two-input multiplexers in a circuit implementing n rotations. For a given n, N_{MUX} has the following bounds:

- N_{MUX} is $O(n)$. Any set of rotations can be implemented using an n-input multiplexer, which in turn is a tree of n 2-input multiplexers.
- N_{MUX} is $\Omega(\log n)$. This is the minimum number of bits specifying one number out of n.

Note that in both cases, the critical path length is logarithmic.

A circuit intermediate between these two extremes can be represented as a network of 2-input multiplexers. We have proven that:

- unless there are distinct indices a, b, c, d such that

$$s_a - s_b = s_c - s_d,$$

the circuit optimal in area and critical path is an n-input multiplexer;

- the original set of rotations k_i does not have the above property.

These facts lead us to the tentative assumption that the solution of the space domain problem can be approximated as an n-input multiplexer. The overall quality measure is then

$$Q = \left(nA_{\mathrm{MUX}}(n) + A_0\right) \cdot \left(\sum_{i=1}^{r} \sum_{j=1}^{n} T_{ij} + T_0 \right) \cdot \max\{\tau_{\mathrm{MUX}}(n), \tau_0\}.$$

The area $A_{\mathrm{MUX}}(n)$ and critical path $\tau_{\mathrm{MUX}}(n)$ of an n-input multiplexer can be expressed in terms of primitive gates, if such a measure is used for the rest of the unit. Alternatively, these values can be measured in terms of implementation technology (transistors, programmable blocks) and as such obtained from the synthesis tools used.

5.3 Time domain problem

Due to the modularity of Eq. 6, the value of all s_j and T_{ij} can be restricted to $(0, m)$ without loss of generality. This still represents, however, a large solution space. To reduce it, further decomposition is used:

1. Choose a set of rotations s_j.
2. Obtain a set of factors T_{ij} giving optimum quality Q.

Because no reliable estimate can be made in Step 1, both steps are performed iteratively. In other words, we use Step 2 as the evaluation function for the local search in Step 1.

5.4 Optimal factors by dynamic programming

In Step 2 above, n is fixed, and so are the parameters of the multiplexer implementing s_j. Furthermore, note that the equations for different values of j in Eq. 6 are *independent*, that is, we have r primitive problems of the following form.

For a given $c \in \mathbf{Z}^+$, $c < m$ and $h \in \mathbf{Z}^+$, find

$$T_{ij} \in \mathbf{Z}^+,\ j = 1 \ldots h$$

such that

$$\sum_{j=1}^{h} s_j T_j \equiv c \bmod m$$

while minimizing

$$q = \sum_{j=1}^{n} T_j .$$

Let $F(h, c)$ be a function giving minimum q for the above problem. Let $F(0, 0) = 0$ and $F(0, c) = \infty$ for $c > 0$. Then $F(h, c)$ is, for $0 < h \le r$,

$$F(h, c) = \min_{d} \left\{ F\left(h-1, (c - d s_h) \bmod m\right) + d \right\},$$

where d ranges between 0 and the order of s_h in $\mathbf{Z}_m$, that is

$$0 \le d \le m / \gcd(s_h, m)$$

The values of $F(h, c)$ are computed for increasing h and are stored in a two-dimensional array with r columns and m rows. After that, the element (r, k_i) contains the contribution of the i-th primitive problem to the optimization criterion, for $1 < i \le r$.

Eventually, all values T_{ij} and therefore the solution of Step 2 can be reconstructed from the array. This means that one pass only of the above outlined dynamic programming procedure is required to solve the entire Step 2. As r is $O(\log m)$, the complexity of this algorithm is $O(m^2 \log m)$.

5.5 Optimal rotation set by a genetic algorithm

The search process in Step 1 is performed by a genetic algorithm, which in turn uses the above procedure to evaluate the solution.

Thanks to the decomposition in Subsection 5.1, a configuration of the search problem is determined by the value of $n \le r$ variables $s_i \in (0, m)$. This is the phenotype of the genetic algorithm.

The genotype (chromosome) is a fixed structure of r variables from $\langle 0, m)$. A zero value is omitted from the phenotype, and equal values of two or more variables in the genotype constitute a single shift in the phenotype. All permutations of the genotype are considered equivalent. This decoder avoids the necessity to store n as a separate variable and the need to work with a variable-length genotype.

Constrained optimization was implemented using a penalty for each rotation k_i, which the individual in question cannot realize. Due to the modularity of Eq. 6, the search space is well connected. The infeasible individuals were not needed to contribute to the connectedness, and therefore the value of the penalty was chosen well above any possible value of Q.

The rest of the genetic algorithm was quite classical, with single-point crossover and linear scaling of fitness values. The adaptive nature of linear scaling caused the convergence to remain unchanged even in the presence of large A_0 and T_0, where the differences in evaluations are relatively small.

It seems that the number of clock cycles spent in iterative squarings is approx. $O\left(k \sqrt[k]{m-1}\right)$, where k is the number of rotation blocks. The total time necessary for the point addition is then

$$T_{\mathrm{PADD}} = O\left(\left(\frac{m}{D} \log m + k \sqrt[k]{m-1}\right) \cdot \log D\right). \quad (7)$$

6 Implementation

The proposed multiplier/inverter has been implemented in the Xilinx Virtex300 FPGA using the Synopsys FPGA Express synthesis tool and the Foundation 3.3i implementation tool.

The functionality has been verified in the ModelSim simulator. Note that point addition and doubling are completely oblivious, i.e. the sequence of steps depends on m only, not on the processed data. Therefore the simulation was also able to show that the numbers of clock cycles used in Equations 3, 4, and 7 were correct.

6.1 Digit-serial multiplier/inverter

Because of limited space, Table 1 presents results for $m = 180$ and for several digit widths D only. No additional blocks performing "long distance" rotations have been used in these cases. As expected, the area of the multiplier/inverter grows with growing D, and can be expressed as approx. $(2 + 0.5\,D) \times m$ slices or $(2 + 0.5\,D)$ slices per 1 bit.

Table 1: Implementation of a modified multiplier/inverter in Xilinx Virtex300

D	Freq (MHz)	#Slices	#Slices per 1 bit	Point addition	
				(clocks)	(μs)
1	122.489	451	2.51	2582	21.08
2	102.533	544	3.02	1412	13.77
3	101.502	632	3.51	1022	10.07
4	100.492	724	4.02	827	8.23
5	97.069	815	4.53	710	7.31
6	94.153	903	5.02	632	6.71
9	64.008	1174	6.52	502	7.84
10	61.054	1265	7.03	476	7.80
12	54.174	1480	8.22	437	8.07
15	58.042	1798	9.99	398	6.86
18	44.607	2026	11.26	372	8.34
20	40.955	2248	12.49	359	8.77

The computation time does not depend on the digit width D in such a straightforward manner. The results in the last column of Table 1 and in Fig. 5 correspond to Eq. 4. The minimum time is obtained for $D = 6$. The other local minima are caused by the granularity of the FPGA. Whenever the capacity of a look-up table is exhausted, the length of the critical path increases.

6.2 Improving the iterative squarings

The optimizer was implemented using the GAlib C++ library [8]. A number of experiments have been performed, with m in the range interesting for elliptic curve cryptography, that is, from 160 to 250. The following facts were observed:

- Where a brute force optimum solution was available, the algorithm gave an identical answer.

- Any realized rotation s_i in an optimum solution was identical to some given rotation k_i, although even slightly sub-optimum solutions did not have this property.
- The left side of Equation 6 was less than m in all optimum and some sub-optimum solutions.
- When the above observation was exploited to simplify the evaluation procedure, the search space became disconnected, and more time was needed to achieve equivalent results.
- Neither brute force nor the described algorithm gave any optimum or sub-optimum solution violating the tentative assumption in Subsection 5.2.
- With a population size of 100, the algorithm required circa 3000 generations to converge at $m = 160$, rising to 4000 at $m = 250$.
- Infeasible individuals were rare.
- The running time was below 20 minutes on an office-grade PC.

Table 2: Shifters adjusted to different multipliers

Multiplier			Shifter		
digit width	A_0	T_0	A	T	rotations s_i
–	0	0	976	10	1, 5, 20, 81
1	489	1956	244	159	1
6	2934	326	488	24	1, 10

Table 2 illustrates the influence of the multiplier size on the shifter. The results were obtained for $m = 163$, where the required set of rotations is {1, 2, 5, 10, 20, 40, 81}.

The area of the multiplier was $A_{\mathrm{MUX}}(n) = 1.5\,n$ and the critical path of the unit was outside the shifter.

The effect of adding one rotation block (i.e. $k = 2$) is illustrated in Fig. 5. The expansion of multiplexers did not influence the clock period, as they did not lie on the critical path.

The new design is systematically faster. For $D = 6$, the speedup is over 20 %, while the area increased by 10 %. Recall that the speedup is based on minimization of the last term in Eq. 7. In our case, this mechanism caused the second local minimum on the original curve to prevail, where the optimum digit width is $D = 15$ for $m = 180$. In this case the actual speedup is 37 %.

7 Conclusions

A pipelined version of the Massey-Omura multiplier modified for easy implementation of ITT inversion algorithm has been presented. The performance of this multiplier/inverter can be improved by employing digit-serialization and by speeding up the iterative squarings.

The multiplier/inverter has been implemented in Xilinx Virtex 300. Without speeding up the iterative squarings, the shortest computation time has been obtained for digit width $D = 6$. The use of "long distance" rotation blocks further speeded up the design and benefited higher digit widths.

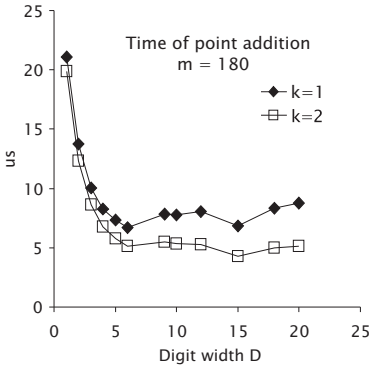

Fig. 5: Time of point addition for different digit widths. One rotation block ($k = 1$) and two rotation blocks ($k = 2$) used

References

[1] Agnew, G. B., Mullin, R. C., Onyszchuk, I. M., Vanstone, S. A.: "An Implementation for a Fast Public-Key Cryptosystem," *Journal of Cryptology*, Vol. **3** (1999), p. 63–79.

[2] IEEE 1363. Standard for Public-key Cryptography, IEEE 2000.

[3] Itoh, T., Teechai, O., Tsujii, S.: "A Fast Algorithm for Computing Multiplicative Inverses in *GF*(2t) Using Normal Bases," *J. Society for Electronic Communications* (Japan), Vol. **44** (1986), p. 31–36.

[4] Leong P. H. W., Leung, K. H.: "A Microcoded Elliptic Curve Processor Using FPGA Technology," IEEE Transactions on VLSI Systems, Vol. **10,** No. 5, Oct. 2002, p. 550–559.

[5] Savas, E., Koc, C. K.: "Architectures for Unified Field Inversion with Applications". In: Elliptic Curve Cryptography. The 9th IEEE International Conference on Electronics, Circuits and Systems – ICECS 2002, Dubrovnik, Croatia, September 15–18, 2002, Vol. **3**, p. 1155–1158.

[6] Massey, J., Omura, J.: "Computational Method and Apparatus for Finite Field Arithmetic," U.S. patent number 4,587,627, 1986.

[7] Blake, I., Seroussi, G., Smart, N.: "Elliptic Curves in Cryptography", Chapter 1. Cambridge University Press, Cambridge (UK), 1999.

[8] Wall, M.: "Galib, A C++ Library of Genetic Algorithm Components" [Online] Available from http://sourceforge.net/projects/galib/

Ing. Jan Schmidt, Ph.D.
phone: +420 224 357 473
e-mail: schmidt@fel.cvut.cz

Ing. Martin Novotný
phone: +420 224 357 261
e-mail: novotnym@fel.cvut.cz

Department of Computer Science and Engineering

Czech Technical University in Prague
Faculty of Eletrical Engineering
Karlovo nám. 13
121 35 Praha 2, Czech Republic

