Efficient Modular Division Implementation
–ECC over GF(p) Affine Coordinates Application–

Guerric Meurice de Dormale, Philippe Bulens, Jean-Jacques Quisquater

Université Catholique de Louvain, UCL Crypto Group,
Laboratoire de Microélectronique (DICE),
Place du Levant 3, B-1348 Louvain-La-Neuve, Belgium.
{gmeurice,bulens,quisquater}@dice.ucl.ac.be

Abstract. Elliptic Curve Public Key Cryptosystems (ECPKC) are becoming increasingly popular for use in mobile appliances where bandwidth and chip area are strongly constrained. For the same level of security, ECPKC use much smaller key length than the commonly used RSA. The underlying operation of affine coordinates elliptic curve point multiplication requires modular multiplication, division/inversion and addition/substraction. To avoid the critical division/inversion operation, other coordinate systems may be chosen, but this implies more operations and a strong increase in memory requirements. So, in area and memory constrained devices, affine coordinates should be preferred, especially over GF(p).
This paper presents a powerful reconfigurable hardware implementation of the Takagi modular divider algorithm. Resulting 256-bit circuits achieved a ratio throughput/area improved by at least 900 % of the only known design in Xilinx Virtex-E technology. Comparison with typical modular multiplication performance is carried out to suggest the use of affine coordinates also for speed reason.

1 Introduction

Modular arithmetic plays an important role in cryptographic systems. In mobile appliances, very efficient implementations are needed to meet the cost constraints while preserving good computing performances. In this field, modular multiplication has received great attention through different proposals: Montgomery multiplication, Quisquater algorithm, Brickell method and some others. The modular inversion problem has also been extensively studied. It can be performed using the well-known Euclid algorithm (or any of the binary variants like the Montgomery inverse [10, 8]), Fermat little theorem or the recently GCD-free method [6].

The modular division is believed to be slow and has not received a lot of attention because it can be replaced by a modular inversion followed by a modular multiplication. Despite their slowness, these operations are needed in several cases: when creating public-private key pairs for RSA and when deciphering in an ElGamal cryptosystem. Although, the main bottleneck arises when we talk about Elliptic Curve Cryptosystems (ECC).

This paper is structured as follows: section 2 reminds the theoretical bases of ECC over GF(p) and the different coordinate systems. We introduce the rewritten algorithm and the reason why it has been chosen in section 3. The main contribution of this paper lies in section 4 where we present our implementation. The opportunity of special adders is discussed and a pipelined architecture is described. Practical results and comparisons with the only known published design are in section 5. A typical modular division design is also introduced, suggesting the use of affine coordinates for their memory and area requirements as much as their computational time. Finally, section 6 concludes the article.

2 Elliptic Curve Operations over GF(p)

An elliptic curve E over $GF(p)$, with p a prime number, is defined as the set of points (x, y) verifying the reduced Weierstraß equation:

$$E : f(X,Y) \triangleq Y^2 - X^3 - aX - b \equiv 0 \bmod p$$

for $a, b \in GF(p)$, each choice of these parameters leading to a different curve. Such a curve is called "non-singular" if its discriminant is different from 0 (this corresponds to three distinct roots). The condition is then rewritten as $4a^3 + 27b^2 \not\equiv 0 \bmod p$.

In ECC, the data to be encrypted is represented by a point P on a chosen curve. The encipherment by the key k is performed by computing $Q = P + P + \cdots + P = kP$. This operation, called scalar multiplication, is usually achieved through the "double and add" method (the adaptation of the well-known "square and multiply" to elliptic curves).

2.1 Point Addition and Doubling in Affine Coordinate

In most cases, the addition of points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ leads to the resulting point $R = (x_R, y_R)$ through the following computation:

$$\begin{cases} x_R = \lambda^2 - x_P - x_Q \bmod p \\ y_R = -y_P + \lambda(x_P - x_R) \bmod p \end{cases} , \lambda = \begin{cases} \frac{y_P - y_Q}{x_P - x_Q} \bmod p & \text{if } P \neq Q \\ \frac{3x_P^2 + a}{2y_P} \bmod p & \text{if } P = Q \end{cases}$$

In some cases, exceptions may arise. When we try to add P to its inverse $-P = (x_P, -y_P \bmod p)$, there is an obvious problem in the computation of λ. This is theoretically tackled by the definition of the point at infinity $\mathcal{O}$. The result is defined as $P + (-P) = \mathcal{O} \Leftrightarrow P + \mathcal{O} = -P$. This means that $\mathcal{O}$ is the identity element. Another issue is the doubling of P when it lies on the x-axis, i. e. when its y-coordinate is 0. As might expected, the result is the point at infinity.

2.2 Using other Coordinate Systems

In the computation of $Q = kP$, the "double and add" method requires on average $(m-1)$ doubling steps and $(m-1)/2$ addition steps according to the Hamming weight of k, where m is the bit length of k. For each of these steps, we need to know the value λ. It can be computed by a modular division or a modular inversion followed by a modular multiplication. As those operations are believed to be slow, one may prefer representing points in another coordinate system [3].

Those kinds of systems allow point addition and point doubling with only multiplications and additions. Inversion is no more intertwined in the elliptic curve operations, but we still need it to convert the final result back to affine representation.

Table 1 contains the timings in modified jacobian[1] and in affine coordinates, where M denotes the time for a modular multiplication, S for a squaring and I for the inversion. $I + M$ can be replaced by a modular division D.

Table 1. Timings of different coordinate systems

	Doubling	Addition
modified jacobian	$t(2\mathcal{J}^m) = 4M + 4S$	$t(\mathcal{J}^m + \mathcal{J}^m) = 13M + 6S$
affine	$t(2\mathcal{A}) = 2M + 2S + I$	$t(\mathcal{A} + \mathcal{A}) = 2M + S + I$

It is now possible to issue the upper bound for the ratio I/M under which the affine coordinates are theoretically more useful. We extract it by asking the average timings for scalar multiplication in affine to be quicker than in jacobian coordinates.

$$\begin{aligned} (m-1) \cdot t(2\mathcal{J}^m) + \tfrac{m-1}{2} \cdot t(\mathcal{J}^m + \mathcal{J}^m) &\geq (m-1) \cdot t(2\mathcal{A}) + \tfrac{m-1}{2} \cdot t(\mathcal{A} + \mathcal{A}) \\ 2 \cdot (4M + 4S) + 13M + 6S &\geq 2 \cdot (2M + 2S + I) + 2M + S + I \\ 8 &\geq I/M \end{aligned}$$

The last result is obtained assuming a square is performed using the same circuitry as a multiplication. The same kind of criteria may be adapted to the ratio D/M. In this case, we reach $9 \geq D/M$. Of course, those timing considerations must be weighted by their area and memory requirements.

3 Algorithm

All the algorithms practically used for modular division are based on the Extended Binary GCD. The main problem is that a comparison is generally needed in order to determine the next operations to compute. For this reason, some efforts have been made to speed up the operation [2], but we believe that the most clever idea is due to the work of Takagi [11], based on the plus-minus algorithm of Brent and Kung [1]. He reaches the goal of avoiding comparison by replacing it with the inspection of Least Significant Bit (LSB) from shift registers and variables.

[1] We choose this system since it is the fastest at point doubling.

Takagi's algorithm is rewritten in algorithm 1. For a m-bit modulus M, it takes between $m+4$ and $2m+4$ clock cycles to perform the loading of the operands, the modular division and the last correction step. The operations between brackets are performed in parallel.

Algorithm 1 Algorithm for modular division computation.

Inputs: $2^{n-1} < M < 2^n$; $-M < X, Y < M$
Output: $Z = X/Y \bmod M$

STEP 1:
  $A \leftarrow 0,\ B \leftarrow Y,\ U \leftarrow 0,\ V \leftarrow X,\ P \leftarrow n,\ D \leftarrow 0$
STEP 1BIS:
  $A \leftarrow A + B,\ B \leftarrow M,\ U \leftarrow U + V,\ V \leftarrow 0$
STEP 2:
  while $P \geq 0$ **do**
    if $[a_1 a_0] = 0$ **then** $A \leftarrow A/4,\ U \leftarrow MQRTR(U, M)$
      if $D < 2$ **then**
        if $D = 1$ **then** $P \leftarrow P - 1$
        else $P \leftarrow P - 2$
      $D \leftarrow D - 2$
    elsif $a_0 = 0$ **then** $A \leftarrow A/2,\ U \leftarrow MHLV(U, M)$
      if $D < 1$ **then** $P \leftarrow P - 1$
      $D \leftarrow D - 1$
    else
      if $([a_1 a_0] + [b_1 b_0]) \bmod 4 = 0$ **then** $q \leftarrow 1$ **else** $q \leftarrow -1$
      if $D \geq 0$ **then** $A \leftarrow (A + qB)/4,\ U \leftarrow MQRTR(U + qV, M)$
        if $D = 0$ **then** $P \leftarrow P - 1$
        $D \leftarrow D - 1$
      else $D \leftarrow -D - 1,\ \{A \leftarrow (A + qB)/4,\ B \leftarrow A\}$
        $\{U \leftarrow MQRTR(U + qV, M),\ V \leftarrow U\}$
STEP 3:
  $U \leftarrow 0$
STEP 4:
  if $[b_1 b_0] \bmod 4 = 3$ **then**
    if $V \geq 0$ **then** $Z \leftarrow U - V + M$ **else** $Z \leftarrow U - V$
  else
    if $V \geq 0$ **then** $Z \leftarrow U + V$ **else** $Z \leftarrow U + V + M$

The presented algorithm is slightly different from the original. The initialization step has been duplicated (STEP 1BIS) and a reset step (STEP 3) has been added before the correction step (STEP 4) to spare resources within the targeted devices. The extra clock cycles added are negligible with respect to the global executing time.

Instead of using redundant binary representation like Takagi, we decided to use classical binary representation. Indeed, we want to focus on the smallest area

requirements and the redundancy roughly twice the amount of hardware needed in the FPGA. With this choice, the timings of the P and D shift registers used in the control part are less critical. So, we decided to replace those registers by small counters, like suggested in [11]. Their sizes are in $\log_2(\log_2(M))$, so the propagation delay is very short. Another fact in aid of counters is that shift registers with different directions and steps consume a lot of resources on FPGA. Finally, counters also enable a slight reduction of the algorithm complexity, leading to a more efficient control structure.

The main feature of this algorithm is the use of P and D counters. The comparison between A and B in commonly used algorithms is replaced by $D = \alpha - \beta$, where α and β are values such that 2^α and 2^β represent the minimums of the upper bounds of $|A|$ and $|B|$ respectively. This substitution reduces the comparison between A and B in the "while" loop to "$|A| > 0$". Instead of investigating all the bits of A, it is replaced by a counter P with sign detection, indicating the minimum of the upper bounds of $|A|$ and $|B|$.

The halving operation $MHLV(T, M)$ of $T \bmod M$ is performed either by $T/2$ or $(T + M)/2$ regarding the parity of T (the LSB). The quartering operation $MQRTR(T, M)$ of $T \bmod M$ depends on the value of $M \bmod 4$. If it equals 1, then the operations to be carried out are $T/4$, $(T - M)/4$, $(T + 2M)/4$ or $(T + M)/4$, accordingly as $T \bmod M$ is $0, 1, 2$ or 3. If $M \bmod 4$ equals 3, the operations are $T/4$, $(T + M)/4$, $(T + 2M)/4$ or $(T - M)/4$ with respect to the value of $T \bmod M$, respectively $0, 1, 2$ or 3.

It should be finally noticed that, when using binary GCD, a division does not slow down the computation compared to the inversion, since U or V are not used in the control part[2].

4 Implementation

In this section, we present two different kinds of implementation. We first introduce the basic sequential architecture and the opportunity of flags precomputation. After, we present an improvement of this architecture by tackling the critical path: the carry chain of the adders. We will show that the best compromise lies in the use of pipelinening instead of carry conditional and select adder.

4.1 Basic Division Architecture

We present here the basic sequential architecture. It is naturally broken up in different distinct parts: the *ControlStage*, the *ABstage*, the *UVstage* and the *P,D* counters.

The ControlStage: The main advantage of the algorithm we use lies in the absence of comparison, serially wired with the main operative part. Fortunately,

[2] U is set to 1 when performing an inversion, to X otherwise.

this improvement implies only a small complexity increase with the use of P and D counters. In the main loop, all the flags only depend on parity bits of variables and on sign of small counters. So, all the flags can be efficiently precomputed, leading to high working frequency.

The ABstage: The ABstage operative part is shown in Fig. 1. It consumes mainly 3 Logic Elements (LE, half of a slice) on FPGA for each bit of the modulus. One is used for the two's complement adder/substractor, another is used for the shift right selection and the last one is used for the loading and the swap of registers.

Since the loading step has been duplicated, we can spare one multiplexor, implemented by one LE per bit length, between the logical shifter and the register.

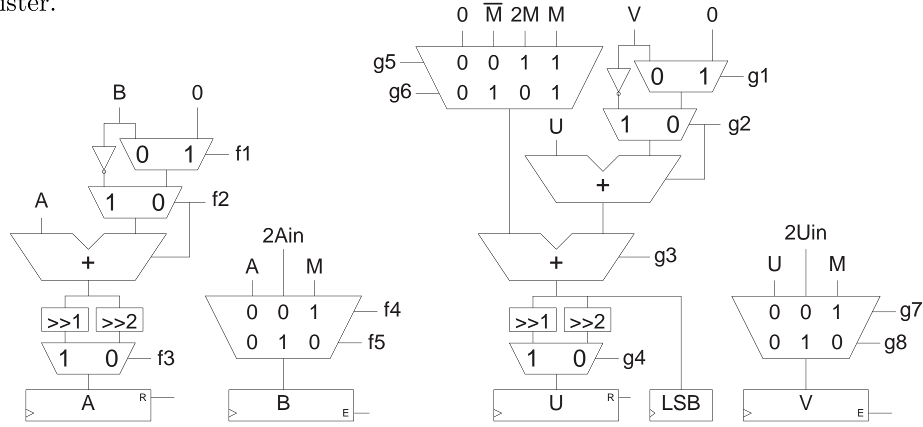

Fig. 1. stage AB

Fig. 2. stage UV

The UVstage: The UVstage operative part is shown in Fig. 2. This stage mainly consumes 4 Logic Elements per bit length. Two are used for two's complement adder/substractor (M is considered constant), one is used for the shift right selection and the last one is used for the loading and the swap of registers.

We also spare a multiplexor thanks to the duplication of the loading step. The reset step (STEP 3) of the U signal enable resource sharing for the last correction step (STEP 4). However, we always right shift the input of the U register. The shifted bit must be saved in one register to provide the final result.

We should notice that the flags $g3$, $g5$ and $g6$ depend on the 2 LSB computed by the first adder. An efficient solution is to replicate this 2-bit adder to allow flags precomputation in the control part.

The *P* and *D* counters: The P counter is based on an adder and is negative allowed (Fig. 3). The end criteria can be checked with only the sign bit. The D counter is also adder based (Fig. 4). We introduced two additional adder which always compute $D-1$ and $D-2$ to compute all the tests needed in the algorithm.

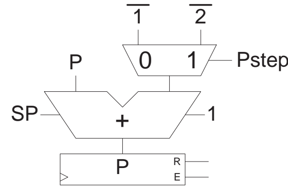

Fig. 3. counter P

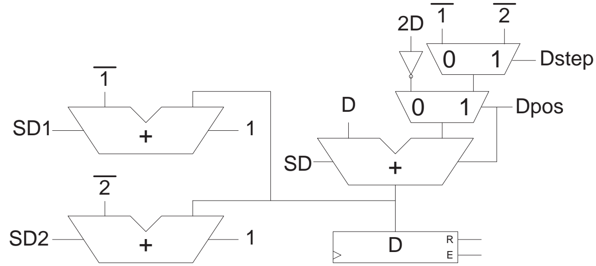

Fig. 4. counter D

4.2 Special Adder

ECC need computation over important length modulus (typically 160 bits). Area constrained modular division designs are adder based, so it is obvious that the critical path resides in the carry chain of the simple carry-propagate adders.

Modern FPGA are built to efficiently implement the addition operation. High speed performance is achieved by the use of a dedicated optimized carry chain. While FPGAs are structured as an array of programmable logic and routing resources, the carry chain is physically wired and therefore exhibits very low delays. It can be used without constraining the routing as long as the carry chain size does not exceed the column height. For this purpose, adders must be adapted to keep satisfactory speed performance.

This optimized carry chain explains why well known methods like carry-look-ahead, carry-bypass, carry conditional, carry-select and carry-save are practically not really attractive on area constrained reconfigurable devices. Of course, some of them can speed up work frequency, but it is at the cost of a lot of chip area.

Another solution is the pipelining of the addition. It is simply achieved by inserting register in the carry chain and by properly handling the operands. This way, almost no additional hardware is consumed. The drawback is the number of clock cycles needed to fill the pipeline. Nevertheless, this overhead can be negligible if the number of repeated additions is great with respect to the number of pipelined stages. These advantages lead us to choose this method for our improved design.

4.3 Pipelined Division Architecture

In order to speed up the computation, we choose the pipelined architecture. As said in the previous section, this method requires small area increase and can lead to interesting results. It can be used because the flags of the control part are only LSB for the main loop. MSB are only required for the last correction step. So, the total overhead will be twice the number of pipeline stages added. This is negligible compared to the number of clock cycles required for the whole computation.

Number of pipeline stages: The number of pipeline stages must still be determined. Putting the overhead aside, we cannot cut the adders in a lot of small parts. This is due to the behaviour of the algorithm: the two right shifts impose that the LSB of the next stage must be available for the MSB of the current stage. Because the computation is achieved in one clock cycle, we must bring back the LSB asynchronously. This increases the number of Logic Elements through the path leading to the computation of the carry-out.

This new design will be quite place & route dependent. So, it is not easy to theoretically determine the best number of pipeline stages. Nevertheless, approximation can be made to decide after how many bits the carry chain must be cut. After inspecting Fig. 5 and Fig. 6, the carry chain propagation time must be shorter than the delay of the U and V operands arrival plus the carry-in arrival and three serial Logic Element (two for the adders and one for the conditional shifter).

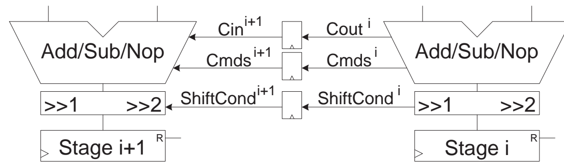

Fig. 5. Typical pipeline organisation

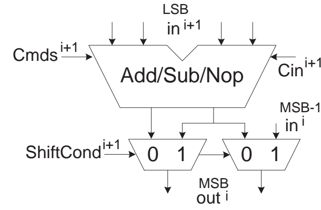

Fig. 6. Modification of the 2 MSB input of each stage

Architecture: To improve readability, all stage features are not reproduced in Fig. 5. The main concern is the conveyance of the shifted bits between pipelined stages. Indeed, the 2 MSB input of each stage (except the last) must be modified with the circuit shown in Fig. 6. The two LSB adders and the conditional shifter have been duplicated to avoid the two serial delays of carry-in arrival and sum return. These two delays appear when we simply take the output from the next pipeline stage adder. With this negligible increase of logic element, we expect to only suffer from the operand arrival delay. All these modifications ensure a faster working design, with only a small increase of hardware and clock cycles.

5 Results

Speed and area comparison of 64, 128, 160 and 256-bit basic and pipelined designs are presented in table 2. The VHDL synthesis and place & route have been achieved on Xilinx ISE 6.2.02i. The first device selected is a Xilinx Virtex-E XCV2000e-6bg560 FPGA (V-E) to guarantee fair comparison with the design [4]. The second is a Xilinx Spartan3 XC3S200-4pq208 (S3) to exhibit performances over a small and low cost FPGA. The two operands are loaded by two 32-bit clocked interface and stored in 32-bit shift registers. The result is registered and then moved in 32-bit shift registers to access the 32-bit clocked output interface.

As explained in section 4.3, the pipelined stages must have a minimal length to be interesting. We decided to pipeline 32-bit adders. The pipelining leads

Table 2. FPGA implementation results

Design	Area V-E (Slices)	Freq. V-E (MHz)	Thr./Area (kbit/s. Slices)	Area S3 (Slices)	Freq. S3 (MHz)	Thr./Area (kbit/s. Slices)
64-bit	420 (2 %)	77	88.8	424 (22 %)	77	88
32-bit $\times 2$	460 (2 %)	83	86.1	461 (23 %)	83	86
128-bit	778 (4 %)	55	34.8	873 (45 %)	48	27
32-bit $\times 4$	842 (4 %)	75	42.8	927 (48 %)	79	41
160-bit	951 (4 %)	45	23.3	1112 (57 %)	44	19.5
32-bit $\times 5$	1022 (5 %)	77	36.3	1180 (61 %)	78	31.8
256-bit	1457 (7 %)	29	9.8	1728 (90 %)	29	7.5
32-bit $\times 8$	1612 (8 %)	77	23	1847 (96 %)	80	20.9

to great improvements, especially for the 256-bit basic design, where the carry chain length is too big for the selected FPGA column height.

Throughput/area ratio comparison between our best design and the best design of [4] is given in table 3. To compute the throughput, we consider the worst case of $(2m-1)$ clock cycles for their design and $(2m+4)+2\,ps$ for our design, where ps is the number of pipeline stages. Unfortunately, they do not give the 160-bit design results.

Table 3. Performance comparisons

Design	Freq (MHz)	Throughput (Mbit/s)	Area (Slices)	Thr./area (kbit/s. Slices)	Improvement
64-bit	45	22.7	1212	18.7	
Our 64-bit	**83**	**39.6**	**460**	**86.1**	**360 %**
128-bit	31	15.6	2215	7	
Our 128-bit	**75**	**36**	**842**	**42.8**	**511 %**
Our 160-bit	**77**	**37**	**1022**	**36.3**	
256-bit	27	13.53	5846	2.3	
Our 256-bit	**77**	**37**	**1612**	**23**	**900 %**

As previously said, it is interesting to compare our modular division implementation with modular multiplication in terms of design performances. We take the 120-bit architecture presented in [5] for its reasonable area requirements.

To enable fair comparison, we implemented our design over the same Virtex1000. Our 128-bit design exhibits a frequency of 74 Mhz and an area of 852 Slices but requires, in the worst case, two times more clock cycles. So, we can

roughly say that, with a multiplier work frequency of 88.5 Mhz and an area of 603 Slices, we reached a ratio D/M of 2.5. This result is more than three times as good as the threshold ratio of section 2.2.

6 Conclusion

A powerful modular division implementation has been presented. It has minimal hardware requirements and high work frequency. In addition to the accurate description of our architecture, we discussed the opportunity to use special adder structures (e.g. carry-save, carry-look-ahead) and exhibit that a pipelined carry-propagate adder seems to be the best choice for an area constrained FPGA implementation of the modular division.

We achieved a throughput/area ratio of 23 kbit/(s.Slices) for a 256-bit design. It represents an improvement by at least 900 % of the only known design in Xilinx Virtex-E technology. Using our implementation, affine coordinates for ECC over GF(p) seem to be attractive for their memory and area requirements but also for speed reason. This suggests the use of our design in Elliptic Curve cryptoprocessor of embedded devices.

References

1. R. P. Brent and H. T. Kung, *Systolic VLSI arrays for linear time GCD computation*, VLSI'83, pages 145-154, 1983.
2. S. Chang-Shantz, *From Euclid's GCD to Montgomery Multiplication to the Great Divide*. Technical report, Sun Microsystems Laboratories TR-2001-95, June 2001.
3. H. Cohen, A. Miyaji and T. Ono, *Efficient Elliptic Curve Exponentiation using Mixed Coordinates*, ASIACRYPT 1998, Springer-Verlag, 1998, LNCS 1423, pp. 51–65.
4. A. Daly, W. Marnane, T. Kerins and E. Popovici, *Fast Modular Division for Application in ECC on Reconfigurable Logic*, The 13th International Conference on Field Programmable Logic and Applications — FPL 2003, Portugal, Lisbon, September 1–3, 2003.
5. A. Daly, W. Marnane, *Efficient architectures for implementing montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic*, Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays, 2002.
6. M. Joye and P. Paillier, *GCD-free Algorithms for Computing Modular Inverses.* Cryptographic Hardware and Embedded Systems — CHES 2003, Springer-Verlag, 2003, LNCS 2779, pp. 243–253.
7. M. E. Kahaira and N. Takagi, *A VLSI Algorithm for Modular Mulitplication/Division*, The 16th IEEE Symposium on Computer Arithmetic — ARITH 16, Spain, Santiago de Compostela, June 15–18, 2003.
8. B. S. Kaliski Jr., *The Montgomery Inverse and its Applications*, IEEE Transactions on Computers, 44(8), pp. 1064–1065, August 1995.
9. D. E. Knuth, *The Art of Computer Programming*, vol. 2 : Seminumerical Algorithms , 2nd ed., Addison-Wesley, 1981.
10. E. Savaş and Ç. K. Koç, *The Montgomery Modular Inverse - Revisited*, IEEE Transactions on Computers, 49(7), pp. 763–766, July 2000.
11. N. Takagi, *A VLSI Algorithm for Modular Division Based on the Binary GCD Algorithm*, IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences, Vol. E81-A, n° 5, pp. 724–728, May 1998.