Two hardware implementations for modular multiplication in the AMNS: Sequential and semi-parallel

Asma Chaouch, Laurent-Stéphane Didier, Fangan Yssouf Dosso, Nadia El Mrabet, Belgacem Bouallegue, Bouraoui Ouni

Version of Record: https://www.sciencedirect.com/science/article/pii/S2214212621000193
Manuscript_bd9355ba6c96247155c8c34b37a5edfd

Two Hardware Implementations for Modular Multiplication in the AMNS: Sequential and Semi-parallel

Asma CHAOUCH[1,3,*], Laurent-Stéphane DIDIER[1], Fangan Yssouf DOSSO[1], Nadia EL MRABET[2], Belgacem BOUALLEGUE[4], Bouraoui OUNI[3]

{didier,dosso,asma-chaouch}@univ-tln.fr

nadia.el-mrabet@emse.fr

{ouni_bouraoui,belgacem_bouallegue}@yahoo.fr

Abstract

We propose two hardware architectures for the modular multiplication over a finite field $\mathbb{F}_p$ using the Adapted Modular Number System (AMNS). We present their optimized implementations, with low latency. Our results are promising as we obtain very competitive timings. According to the size of the modulus p, our fastest multiplication is performed in $0.07\mu s$ for 128 bits, $0.146\mu s$ for 256 bits, and $0.172\mu s$ for 512 bits.

Keywords: Modular multiplication, Hardware implementation, FPGA, Adapted modular number system

1. Introduction

Nowadays, cryptography primitives are implemented for miscellaneous uses and the efficiency of their implementation on small embedded system

*Corresponding author
Email address: asma-chaouch@univ-tln.fr (Asma CHAOUCH)
[1]IMATH, Université de Toulon, France
[2]Mines Saint-Etienne, CEA-Tech, Centre CMP, Departement SAS
[3]EμE, Université de Monastir, Tunisie
[4]College of Computer Science, King Khalid University, Saudi Arabia

devices is important. Most of the public key cryptographic protocols involve modular arithmetic; indeed. This family of cryptosystem which includes RSA Rivest et al. (1978b) and elliptic curve cryptography (ECC) Hankerson and Menezes (2011), relies on it A. Menezes (1997). Hence, the most efficient is the modular arithmetic implementation, the most reduced is the execution time. The most critical operation is the modular multiplication. Many algorithms for the binary representation have been proposed in the literature Barrett (1987); Montgomery (1985); Taylor (1981); Blakely (1983); Takagi (1992).

Beside the algorithmic solutions, it is possible to improve their efficiency by choosing unusual number representations. In Bajard et al. (2004), a new number system called the Adapted Modular Number System (AMNS) is proposed in order to speed-up the modular operations. Its main characteristic is that its elements are polynomials with small coefficients. In Didier et al. (2020) the theoretical background of AMNS present with a software implementation of the modular multiplication in this system. They show how to build an efficient AMNS that allows modular multiplications that can be more efficient than the classical Montgomery modular multiplication.

In this paper, we present a FPGA architecture for modular multiplication using AMNS and analyze its behavior. This paper gives guidelines to the material fitting of such a structure, namely two methods are presented and a suitable material model is identified. In order to evaluate the performances and the costs (in number of slices register and DSP blocks) of our architectures, we compare our results with the state of the art of modular multiplications in binary, signed-digits and RNS representation. Various architectures have been built for the finite field $\mathbb{F}_p$, where p is a prime integer, to provide a scalable implementation on FPGA. Our results are very promising as our implementations provide the most efficient modular multiplication compared to binary implementations with the same size of the modulus p and on the same FPGA target (see Table 8). The work presented is done in continuation of Chaouch et al. (2019).

The paper is organized as follows. In Section 2, we present the background of the modular multiplication and the AMNS. We describe our hardware architectures in section 3 and in section 4. Our results and a comparison with the state of the art are presented in section 5 and in Section 6. We conclude in section 7.

2. The Adapted Modular Number System

The Adapted Modular Number System (AMNS) is a number system that was introduced in Bajard et al. (2004), in order to speed-up modular arithmetic. The main characteristic of the AMNS is that its elements are polynomials. This characteristic gives to the AMNS many advantages for both efficiency and safety. In Didier et al. (2020), the authors give a method to generate many AMNS for any prime integer in order to perform modular operations efficiently. They present software implementation results which highlight that the AMNS allows to perform modular multiplication more efficiently than well known libraries like GNU-MP and OpenSSL. In this section, we give an overview of this number system and of modular operations in it.

In this paper, we do not deal with the generation process of the AMNS, since it requires a consequent mathematical background that is out of the scope of this article and is quite long. See Bajard et al. (2004); Didier et al. (2020). The authors of Didier et al. (2020) provide an implementation of the AMNS generation process [5] that uses SageMath library Stein and al. (2018). In the same location, there is also a C code generator. Given all the parameters of an AMNS, this generator outputs a software implementation in C language that allows to efficiently perform arithmetic operations in that AMNS.

The AMNS is a subclass of the Modular Number System (MNS). So, we start by first presenting the MNS.

Definition 1. Let $p \geqslant 3$ be a prime integer. A modular number system (MNS) is defined by a tuple $\mathcal{B} = (p, n, \gamma, \rho)$, such that for every integer $0 \leqslant x < p$, there exists a vector $V = (v_0, \ldots, v_{n-1})$ such that:

$$x \equiv \sum_{i=0}^{n-1} v_i \gamma^i \pmod{p},$$

with $|v_i| < \rho, \quad \rho \approx p^{1/n}, \quad \text{and} \quad 0 < \gamma < p.$

In this case, we say that the polynomial $V(X) = v_0 + v_1 X + \cdots + v_{n-1} X^{n-1}$ is a representation of x in $\mathcal{B}$ and we denote $V \equiv x_{\mathcal{B}}$.

For a MNS to be an AMNS, the parameter γ should meet a requirement which is essential for arithmetic operations.

[5] `https://github.com/arithPMNS/generalisation_amns`

Definition 2. An Adapted Modular Number System (AMNS) is defined by a tuple $\mathcal{B} = (p, n, \gamma, \rho, E)$ such that (p, n, γ, ρ) is a MNS and γ is a root modulo p of the polynomial $E(X) = X^n - \lambda$, with λ a very small nonzero integer (for instance, $\lambda = \pm 1$, ± 2 or ± 3).

Example 1. Let $p = 19$. The tuple $\mathcal{B} = (19, 3, 7, 2, E)$, with $E(X) = X^3 - 1$, is an AMNS. Indeed, Table 1 gives a representation of each element of $\mathbb{Z}/19\mathbb{Z}$. This shows that the tuple $\mathcal{B} = (19, 3, 7, 2)$ is a MNS. Moreover, we have $\gamma^n = 7^3 \equiv 1 \pmod{19}$, which is very small.

0	1	2	3	4
0	1	$-X^2 - X + 1$	$X^2 - X - 1$	$X^2 - X$

5	6	7	8	9
$X^2 - X + 1$	$X - 1$	X	$X + 1$	$-X^2 + 1$

10	11	12	13	14
$X^2 - 1$	X^2	$X^2 + 1$	$-X + 1$	$-X^2 + X - 1$

15	16	17	18
$-X^2 + X$	$-X^2 + X + 1$	$X^2 + X - 1$	-1

Table 1: The elements of $\mathbb{Z}/19\mathbb{Z}$ in $\mathcal{B} = (19, 3, 7, 2)$

It can be checked in Table 1 that any representation A of an element $a \in \mathbb{Z}/19\mathbb{Z}$ is such that: $\deg(A) < 3$, $\|A\|_\infty < 2$ and $A(\gamma) \equiv a \pmod{p}$. For instance, $\gamma^2 - \gamma + 1 = 49 - 7 + 1 = 43 \equiv 5 \bmod 19$ shows that $X^2 - X + 1$ is a representation of 5 in $\mathcal{B}$.

Like in usual number systems, the main arithmetic operations in the AMNS are the addition and the multiplication. Since elements are polynomials in the AMNS, the addition (resp. the multiplication) is an addition (resp. the multiplication) of polynomials. However, additional operations have to be done in order to obtain the result in the AMNS. This is explained by the following:

Let $x, y \in \mathbb{Z}/p\mathbb{Z}$ be two integers. Let $V \equiv x_{\mathcal{B}}$ and $W \equiv y_{\mathcal{B}}$ be their representations in $\mathcal{B}$. The polynomial $T = VW$ satisfies $T(\gamma) \equiv xy \pmod{p}$. However, T might not be a valid representation of xy in $\mathcal{B}$, because its degree could be greater than or equal to n. To keep the degree lower than n, the product VW has to be computed modulo the polynomial E. This operation is called the *external reduction*. Notice that since $E(\gamma) \equiv 0 \pmod{p}$ and $T = VW \pmod{E}$, we have $T(\gamma) \equiv xy \pmod{p}$ and $\deg(T) < n$.

Even if $\deg(T) < n$, T might not be a representation of $xy \pmod{p}$ in $\mathcal{B}$, because its coefficients could be greater than or equal to ρ. To have the result in $\mathcal{B}$, a specific primitive called the *internal reduction* has to be applied.

The same reasoning can be applied for the addition, $S = V + W$ satisfies $S(\gamma) \equiv (x + y) \pmod{p}$ and $\deg(S) < n$. Here also, S might not be a valid representation in $\mathcal{B}$, since its coefficients could be greater than or equal to ρ. Hence, the same *internal reduction* might be required to retrieve the result in $\mathcal{B}$.

2.1. Some notations and conventions.

Before presenting the arithmetic operations with the reduction methods, we need to establish some notations and conventions for simplicity and consistency. For consistency, we assume that $p \geqslant 3$ and $n, \gamma, \rho \geqslant 1$.

Let $\mathbb{Z}_n[X]$ be the set of polynomials in $\mathbb{Z}[X]$ of degree smaller than n:

$$\mathbb{Z}_n[X] = \{C \in \mathbb{Z}[X], \text{ such that: } \deg(C) < n\}.$$

If $V \in \mathbb{Z}_n[X]$ is a polynomial, we assume that $V(X) = v_0 + v_1 X + \cdots + v_{n-1} X^{n-1}$.

2.2. The external reduction.

The *external reduction* is a polynomial modular reduction. The goal of this operation is to keep the degree of the AMNS representations lower than n. Let $C \in \mathbb{Z}[X]$ be a polynomial. The external reduction consists in computing a polynomial C' such that:

$$C' \in \mathbb{Z}_n[X] \text{ and } C'(\gamma) \equiv C(\gamma) \pmod{p}.$$

The Euclidean division of C by E computes Q and C' so that:

$$C = Q \times E + C',$$

with $\deg(C') < n$ and $Q \in \mathbb{Z}[X]$. Since $E(\gamma) \equiv 0 \pmod{p}$, one has $C'(\gamma) \equiv C(\gamma) \pmod{p}$. So, the external reduction is done as: $C' = C \bmod E$. The polynomial E is called the *external reduction polynomial*.

Let $A \in \mathbb{Z}_n[X]$ and $B \in \mathbb{Z}_n[X]$. Let $C = AB$ be a polynomial. Then, $\deg(C) < 2n - 1$. Since $E(X) = X^n - \lambda$, with λ very small, the external reduction can be done very efficiently. The function RedExt (Algorithm 1) proposed by Plantard in Plantard (2005) can be used to perform this operation.

Algorithm 1 RedExt - External reduction Plantard (2005)

Require: $C \in \mathbb{Z}[X]$ with $\deg(C) < 2n - 1$ and $E(X) = X^n - \lambda$

Ensure: $C' \in \mathbb{Z}_n[X]$, such that $C' = C \bmod E$

1: **for** $i = 0 \dots n - 2$ **do**
2: $\quad c'_i \leftarrow c_i + \lambda c_{n+i}$
3: **end for**
4: $c'_{n-1} \leftarrow c_{n-1}$
5: return C' $\quad \# \ C' = (c'_0, \ \dots, \ c'_{n-1})$

Remark 1. In Bajard et al. (2004), the authors show that the output C' of Algorithm 1 is such that:

$$\|C'\|_\infty < n|\lambda|\|A\|_\infty\|B\|_\infty,$$

with $C = AB$ and $A, B \in \mathbb{Z}_n[X]$; cf. Section 3 of that paper.

2.3. The internal reduction.

The aim of the *internal reduction* is to ensure that the coefficients of polynomials are lower (in absolute value) than ρ. Let $C' \in \mathbb{Z}_n[X]$ be a polynomial, with $\|C'\|_\infty \geqslant \rho$. This operation consists in computing a polynomial S such that $S \in \mathcal{B}$ and $S(\gamma) \equiv C'(\gamma) \pmod{p}$.

Several methods have been proposed to perform this operation Bajard et al. (2004, 2005); Negre and Plantard (2008). If the value of the modulus p is not imposed but only its approximate bit-size, then the proposal in Bajard et al. (2004) might achieve the best efficiency. However, if the value of p is already given, then the proposal in Negre and Plantard (2008) is currently the best and is the one this paper focuses on.

In Negre and Plantard (2008), Negre and Plantard propose a Montgomery-like reduction method to perform the internal reduction. In their proposal, they combine the multiplication operation with their Montgomery-like internal reduction. In this paper, we split the multiplication process from the internal reduction, because the internal reduction is also essential for other operations like the addition and the conversions from $\mathbb{Z}/p\mathbb{Z}$ to the AMNS (and vice versa). RedCoeff (Algorithm 2) corresponds to the internal reduction process included in Algorithm 1 of Negre and Plantard (2008).

Algorithm 2 RedCoeff - Coefficient reduction Negre and Plantard (2008)

Require: $\mathcal{B} = (p, n, \gamma, \rho, E)$, $C' \in \mathbb{Z}_n[X]$, $M \in \mathcal{B}$ such that $M(\gamma) \equiv 0 \pmod{p}$, $\phi \in \mathbb{N} \setminus \{0\}$ and $M' = -M^{-1} \bmod (E, \phi)$.

Ensure: $S(\gamma) = C'(\gamma)\phi^{-1} \pmod{p}$, with $S \in \mathbb{Z}_n[X]$

1: $Q \leftarrow C' \times M' \bmod (E, \phi)$

2: $R \leftarrow (Q \times M) \bmod E$

3: $S \leftarrow (C' + R)/\phi$

4: return S

Notice that this Montgomery-like reduction method requires three additional parameters: a non-zero integer ϕ and two polynomials M and M'. These polynomials are such that: $M \in \mathcal{B}$, $M(\gamma) \equiv 0 \pmod{p}$ and $M' = -M^{-1} \bmod (E, \phi)$. The polynomial M is called the *internal reduction polynomial*.

Since this algorithm involves many reductions modulo ϕ (line 1) and many exact divisions by ϕ (line 3), it is necessary to take ϕ being power of two for this algorithm to be efficient. However, taking ϕ as a power of two while ensuring the existence of the polynomials M and M' is not obvious. In Negre and Plantard (2008), the authors do not provide a generation process which allows that. In El Mrabet and Gama (2012), El Mrabet and Gama give such a generation process but for the special case $E(X) = X^n + 1$.

In Didier et al. (2020), the authors propose a generation process which ensures this requirement for $E(X) = X^n - \lambda$, with $\lambda \in \mathbb{Z} \setminus \{0\}$. In the same paper, they also give a generation process for all the others parameters along with a complete set of algorithms to perform arithmetic and conversion operations in the AMNS. Here, we only present the multiplication algorithm since it is the purpose of this paper. One should read the works done in Didier et al. (2020) for more details.

2.4. The modular multiplication.

As explained above, the authors in Negre and Plantard (2008) suggest a modular multiplication method which includes the Montgomery-like reduction method. Algorithm 3 corresponds to their proposal, except that it is split here in a set of algorithms.

Algorithm 3 Modular multiplication in AMNS Negre and Plantard (2008)

Require: $A \in \mathcal{B}$, $B \in \mathcal{B}$ and $\mathcal{B} = (p,\, n,\, \gamma,\, \rho, E)$

Ensure: $S \in \mathcal{B}$ with $S(\gamma) \equiv A(\gamma)B(\gamma)\phi^{-1} \pmod{p}$

1: $C \leftarrow A \times B$

2: $C' \leftarrow \mathrm{RedExt}(C)$

3: $S \leftarrow \mathrm{RedCoeff}(C')$

4: return S

In Negre and Plantard (2008), Negre and Plantard show that if the parameters ρ and ϕ are such that:

$$\rho \geqslant 2n|\lambda|\|M\|_\infty \text{ and } \phi \geqslant 2n|\lambda|\rho,$$

then, the output S of the Algorithm 3 is such that $\|S\|_\infty < \rho$ (i.e. $S \in \mathcal{B}$); cf. Theorem 1 in Negre and Plantard (2008). This requirement is assumed for ϕ and ρ in the Algorithm 3.

In Didier et al. (2020), the authors prove that it is always possible to generate an AMNS for any prime integer $p \geqslant 3$ where the parameter E is such that $E(X) = X^n - \lambda$, with $\lambda = \pm 2^i$. This makes the internal and the external reductions faster since these operations involve many multiplications by λ. As already mentioned, ϕ should be a power of two; i.e. $\phi = 2^t$, with $t \in \mathbb{N} \setminus \{0\}$.

Let's consider a k-bit processor architecture, then the basic arithmetic computations are performed on k-bit words. The cost of the modular multiplication is given in Table 2 with $\lambda = \pm 2^i$ and $\phi = 2^t$. This cost is expressed as a function of the number of k-bit integer multiplications, additions and shifts. In that table, $\mathcal{M}$ and $\mathcal{A}$ respectively denote the multiplication and the addition of two k-bit integers. We also denote respectively $\mathcal{S}_l^y$ and $\mathcal{S}_r^y$ a left shift and a right shift of y bits.

Simple polynomial multiplication	$n^2\mathcal{M} + (2n^2 - 4n + 2)\mathcal{A}$
External reduction	$2(n-1)\mathcal{A} + (n-1)\mathcal{S}_l^i$
Internal reduction	$2n^2\mathcal{M} + (3n^2 - n)\mathcal{A} + n\mathcal{S}_r^t$
Total cost	$3n^2\mathcal{M} + (5n^2 - 3n)\mathcal{A} + (n-1)\mathcal{S}_l^i + n\mathcal{S}_r^t$

Table 2: Theoretical cost of the modular multiplication, where $E(X) = X^n \pm 2^i$ and $\phi = 2^t$.

2.5. Some advantages of the AMNS.

Compared to the classical (binary) representation, the AMNS has many advantages. First, none of the algorithms for arithmetic operations in the AMNS has a conditional branching. This property is an advantage for efficiency and is also very helpful against side channel attacks, like the Simple Power Analysis (SPA) attack. Second, because the AMNS elements are polynomials, their coefficients are independent and there is no carry propagation to deal with, when performing arithmetic operations.

Finally, because its elements are polynomials, arithmetic operations in the AMNS can be very efficiently parallelized.

3. Methodologies for hardware implementation

In our modular multiplication we use the AMNS (Adapted modular number system) algorithm. Due to its regularity and simplicity, this algorithms can be easily implemented in the DSP slices of FPGAs.

Most of the proposed solutions in the literature use complex and expensive algorithms to release these data dependencies and improve computing performance in multipliers implanted. Due to the complexity of these algorithms, the corresponding fast modular multipliers often consume a significant amount of FPGA hardware resources. Thus, we can find in the literature examples of FPGA implementations of modular multipliers within ECC cryptosystems using several hundred DSP slices (see for example H. Alrimeih (2014)). We believe that such a consumption of resources is not reasonable for material implementations of embedded cryptosystems. We have therefore decided to propose a different solution based on architectural optimization rather than algorithms.

Our constraint is to implement a hight level security of cryptographic (128 or 256, 512 and 1024) in FPGA, knowing that the hardwired control units are not big enough for cryptographic sizes. In our target FPGAs, DSP slices are 18×25 bits units. Then processing each $\mathbb{F}_p$ element would require many parallel BRAMs and DSP slices. But using many BRAMs is useless since the number of intermediate $\mathbb{F}_p$ elements is small in ECC and HECC (up to 10 and 20 respectively). To allow hardware design having an efficient area and frequency on FPGA, we decompose k-bit elements into a a set of w-bit words so that w is a multiple of 18. The choice can reduce the number of slice DSP used in the design. We denote n the number of coefficient of w-bit necessary for each operand with $k = nw$.

3.1. The system model

Our hardware implementation of the modular multiplication in AMNS is based on the algorithm 3. This algorithm can be expanded as follows:

Algorithm 4 Modular multiplication in AMNS - Main steps

Require: $A \in \mathcal{B}$, $B \in \mathcal{B}$ and $\mathcal{B} = (p,\, n,\, \gamma,\, \rho, E)$

Ensure: $S \in \mathcal{B}$

$$C \leftarrow A \times B \tag{1}$$
$$C' \leftarrow C \bmod E \tag{2}$$
$$Q \leftarrow C' \times M' \bmod (E, 2^r) \tag{3}$$
$$R \leftarrow ((Q \times M)) \bmod E \tag{4}$$
$$S \leftarrow (C' + R)/2^r \tag{5}$$

We remind that the inputs A, B and C are polynomials of degree $n-1$. Thus, the polynomial A has n coefficients noted a_i for $i \in \{0, \cdots, n\}$. The polynomial M is the internal reduction polynomial and the modular inverse of its opposite modulo $(E, 2^r)$ is M'. Note that for efficiency reasons, we choose $\phi = 2^r$. The external reduction polynomial E is a sparse polynomial of the form $E(X) = X^n - \lambda$. The polynomials C, C' and R are intermediate polynomials. This algorithm produces the output polynomial S being the result of the modular multiplication.

The coefficients of the inputs A, B, C and the constants M and M' are stored in the memory. The result is written back in the same memory. The structure of our architecture is given in Fig. 1. In this figure, the *AMNS*

Multiple-Coefficient operator, processes the steps (1), (2), (3), (4) and (5) of the AMNS multiplication algorithm. It reads the coefficient of each input value a_i, b_j, m'_j and m_j out of the memory, processes the intermediate results C', Q, R, S and store them in dedicated local registers. This allows C', Q and R to be used in the *AMNS Multiple-Coefficient* operator as intermediate values that are input for the next cycle. The register dedicated to S supplies the result to the memory. This unit is controlled with a finite state machine.

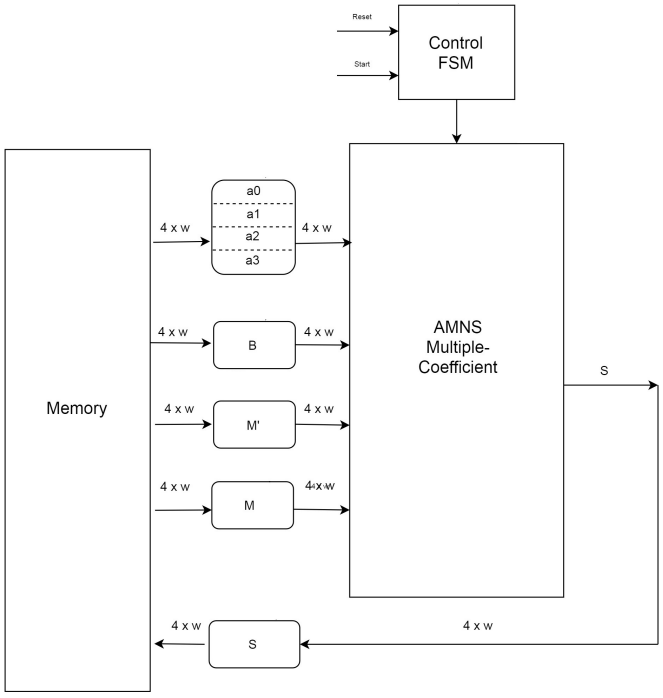

Figure 1: System architecture

The target devices have seldom enough input pins in order to get cryptographic-size data in one single cycle. For instance the board used in our tests has only 220 I/O pin while the modular multiplication data-path is designed up to 1024-bit. Therefore, the inputs of the top level of the design are partitioned in 36-bit chunks and serially delivered with the least significant bits first.

We detail the several steps processed in the *AMNS Multiple-Coefficient* unit for $n = 4$.

We can directly computes $C' = A \times B \bmod E$ as follows:

- $c'_0 = a_0b_0 + \lambda(a_1b_3 + a_2b_2 + a_3b_1)$,
- $c'_1 = a_0b_1 + a_1b_0 + \lambda(a_2b_3 + a_3b_2)$,
- $c'_2 = a_0b_2 + a_1b_1 + a_2b_0 + \lambda(a_3b_3)$,
- $c'_3 = a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0$.

Note that this step combine the polynomial multiplication (1) and the polynomial reduction (2). Indeed, the external reduction polynomial $E = X^n - \lambda$ is a sparse polynomial which property is $X^n \bmod E = \lambda$. Thus the product $C = A \times B$ and the reduction $C' = C \bmod E$ can be combine. The polynomial multiplication is usually implemented using a multiplier-accumulator (MAC operator) with a fixed operation data width (e.g., 36 bits).

The second step computes $Q = C' \times M' \bmod (E, 2^r)$ as follows :

- $Q_0 = c'_0m'_0 + c'_1m'_3 + c'_2m'_2 + c'_3m'_1$,
- $Q_1 = c'_0m'_1 + c'_1m'_0 + c'_2m'_3 + c'_3m_2$,
- $Q_2 = c'_0m'_2 + c'_1m'_1 + c'_2m'_0 + c'_3m_3$,
- $Q_3 = c'_0m'_3 + c'_1m'_2 + c'_2m'_1 + c'_3m'_0$.

The third step computes $R = ((Q \times M)) \bmod E$:

- $R_0 = q_0m_0 + q_1m_3 + q_2m_2 + q_3m_1$,
- $R_1 = q_0m_1 + q_1m_0 + q_2m_3 + q_3m_2$,
- $R_2 = q_0m_2 + q_1m_1 + q_2m_0 + q_3m_3$,
- $R_3 = q_0m_3 + q_1m_2 + q_2m_1 + q_3m_0$.

The last step is simply $S = (C' + R)/2^r$. Thus the modular multiplication is processed through four states, such that the final result S is available at state five. This is illustrated in Fig. 2

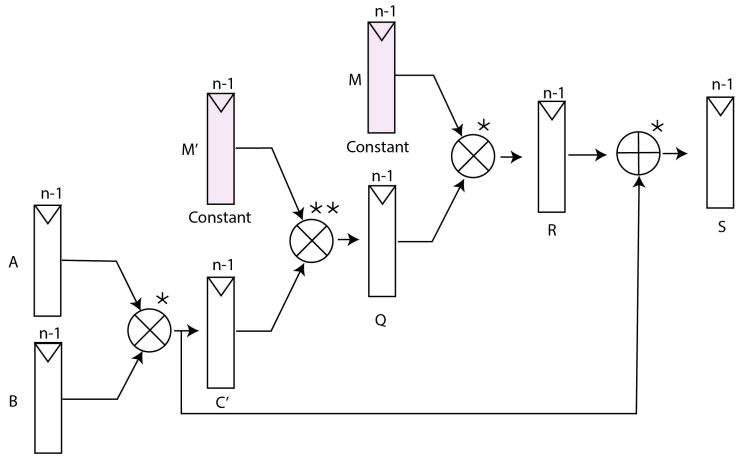

Figure 2: Illustration of modular multiplication operations in AMNS. The notation "*" represents $\bmod(E)$, and "**" represents $\bmod(E, \phi)$

4. Hardware Implementation

In this section, we present two architectures that can be used for computing a modular multiplication in the AMNS on FPGA.

The first is a sequential architecture, which is easy to realize because the arithmetic operations are performed in the order described in Fig. 3. A preliminary analysis shows that the sequential architecture uses fewer slices registers than the semi-parallel architecture but requires more clock cycles for completing the modular multiplication in the AMNS. The second architecture is semi−parallel architecture which aims to execute the most arithmetic operations in the same cycle to have a fast design.

In Section 4.1, we present our sequential architecture while our semi-parallel architecture is presented presented in section 4.2.

4.1. Sequential architecture

In this design the steps described in section 3.1 are scheduled as shown in figure 3. This figure is an example of the scheduling of a modular multiplication in AMNS, which inputs A and B have four 36−bits coefficients and the modulus p is a 128-bit integer.

In figure 1 the *AMNS Multiple-Coefficient* block executes the foregoing algorithm as follows. It inputs the loop variables i and j from the control state machine and supplies them to the memory as its address input. The

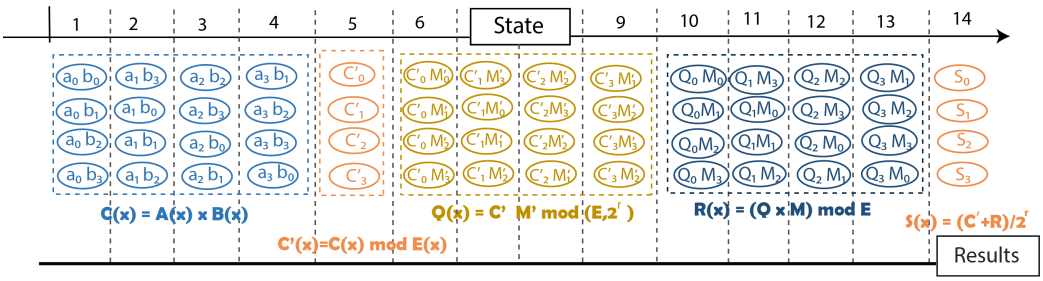

Figure 3: Scheduling of the sequential architecture

input values a_i, b_j, m'_j, and m_j are arrays of w-bits coefficients. Here $w = 36$-bits for a_i, b_j, $w = 39$-bits for m'_j and $w = 40$-bits for m_j.

The *AMNS Multiple-Coefficient* block computes the 5 steps of algorithm4. The step (1) calculates an intermediate variable C Step by Step, from the least significant coefficient to the most significant coefficient. At each cycle a new value of c_0, c_1, c_2 and c_3 comes out as is illustrated in the Fig. 2.

The process executes step (2) computes another intermediate variable C' similarly to step (1). The input C is given serially least-significant coefficient first in order to compute the least significant coefficient of C'. The computation of steps (3) and (4) that compute the intermediate variables Q and R are performed the same way. Finally, the division in step (5) is performed by a simple r-bit right shift. The final result is computed at this step.

The steps (1), (3) and (4) requires n states in order to be computed, while the steps (2) and (5) are performed in one state.

The Fig. 4, Fig. 5, Fig. 6 and Fig. 7 shows the parts of the *AMNS Multiple-Coefficient* block that computes steps (1) and (2), (3), (4) and (5). While these figures do not show it, the AMNS Multiplication architecture includes a $36 - bit$ wide single port memory, which allows either read or write access to a single address in one clock cycle. This memory stores the input values A, B, M and M' for computation as well as receives the output value S. It also serves as temporary storage for intermediate variables C', R and Q.

The computation of step (1) and (2) is illustrated in figure 4. Each 72-bit coefficient of C' is computed with a 36-bit inputs MAC operator. For instance, for a 128-bit precision architecture, four MAC operators are required at this step. Such a data-path requires 3 DSP blocks for each MAC operator. Each MAC performs a multiplication of coefficients of A and B

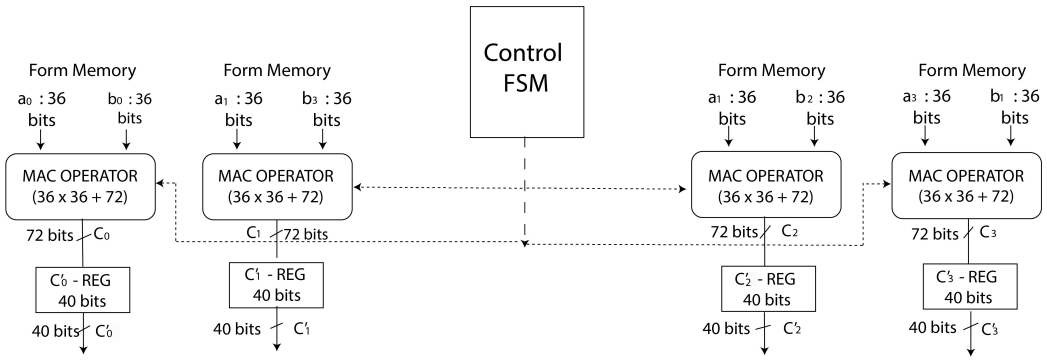

Figure 4: Sequential architecture, computation of step (1) and (2)

taken from memory and accumulates the result. The final value is stored in the C'_i registers that can output serially the result. It has to be remarked that the step (3) which used C' has to be computed modulo(E,ϕ). Thus, in this example, only 40 bits of the coefficients c'_i are needed at the next step.

The architecture of the module that computes the step (3) of the modular multiplication is similar to the previous module except the size of inputs and outputs (Fig. 5). In this module, each MAC operator inputs one 39-bit coefficient of M' and one 40-bit coefficient of C'. Each MAC operator outputs one 79-bit coefficient of Q, but only 40 bits are kept because step (3) is computed modulo(E,ϕ).

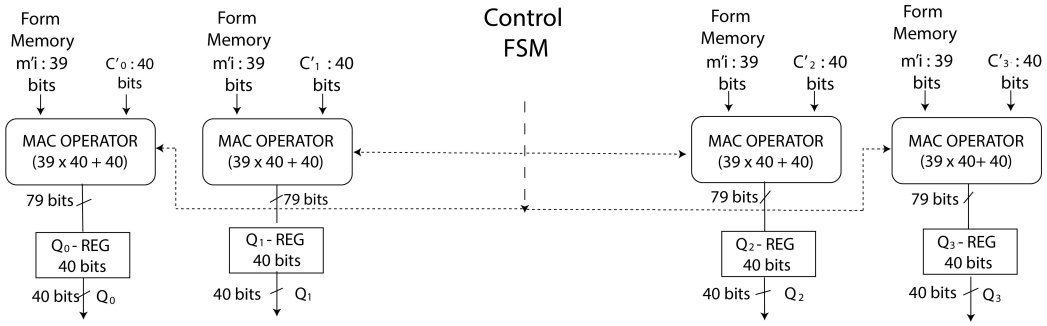

Figure 5: Sequential architecture, computation of step (3)

The computation of step (4) is illustrated by figure 6. This module is similar to the module displayed in figure 4 except the size of the inputs

and the output. Each MAC operator inputs a 36-bit coefficient of M and a coefficient of Q that is serially given. Each module outputs a 76-bit coefficient of R which is stored in a 76-bit register (not displayed in Fig. 6).

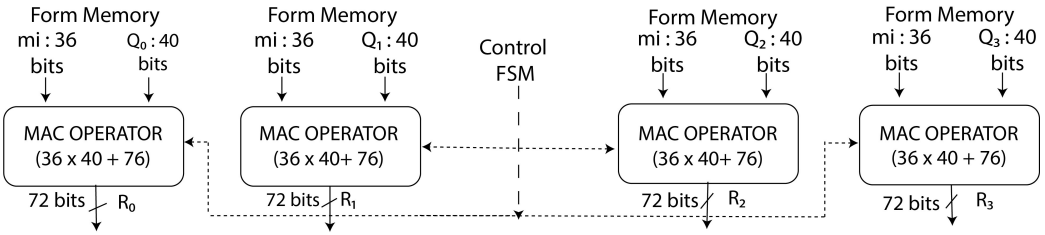

Figure 6: Sequential architecture, computation of step (4)

The final step (5) consists of a 76-bit addition and a shift. The module that computes this step is illustrated in figure 7. Each adder inputs a 72-bit coefficient of C' and a 76-bit coefficient of R. The following shift operator corresponds to the division in step (5). In this example the divisor is 2^{40}. The 36 bit coefficient of the result S are stored back to the memory.

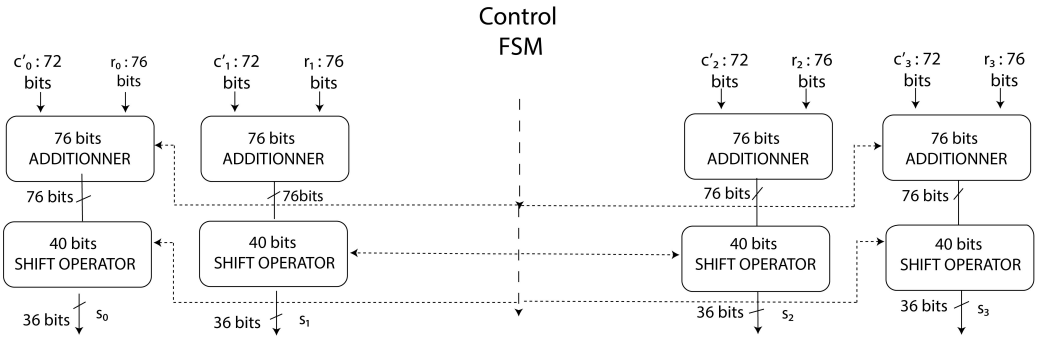

Figure 7: Sequential architecture, computation of step (5)

Main controller (MC). The controller interface signals are set according to the synchronous finite state machine (FSM). This FSM is illustrated in Fig 8 for a 128-bit modulus p, where in state $\mathcal{S}_0$ is the initial state. Each state of

our FSM requires two cycles to be completed, except the final state which is executed in only one cycle. The controller has 14 states. Our basic sequential architecture requires a total of 28 cycles for computing a modular multiplication for a 128-bit modulus p and 36 bits-coefficients AMNS numbers.

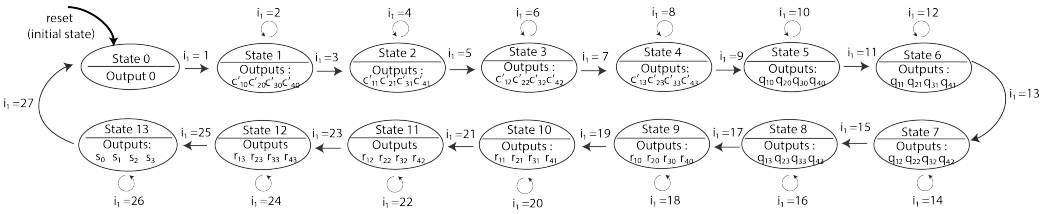

Figure 8: Finite state machine for $w = 36$ and a 128-bit modulus p - Sequential architecture

The data path consists of eight internal registers, a counter and a comparator. The controller remains in the state $\mathcal{S}_0$ until the START instruction is set. When the START signal is set, the a_i and b_i registers are loaded with input values, the $C_{00}, C_{10}, C_{20}, C_{30}$ registers and the counter are reset. The different states are summarized in table 3. The remaining states work as follows:

- In the state $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$ and $\mathcal{S}_4$ the FSM machine execute the steps (1) and (2) of the algorithm.
- $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$ and $\mathcal{S}_4$ are waiting to validate the $DONE\_C'$ signal. Once the coefficients of the polynomial C' are charged in the BRAM register, the FSM change to the next state.
- In the state $\mathcal{S}_5$, $\mathcal{S}_6$, $\mathcal{S}_7$ and $\mathcal{S}_8$ are waiting to validate the $DONE\_Q$ signal. They computes the step (3) in the algorithm. Once the coefficients the polynomial Q are charged in the BRAM register, the FSM change to the next state.
- $\mathcal{S}_9$, $\mathcal{S}_{10}$, $\mathcal{S}_{11}$ and $\mathcal{S}_{12}$ control the computation of the step (4) of the algorithm. They are waiting to validate the $DONE\_R$ signal. Once the FSM machine generates the coefficients of the polynomial R and charges them in the BRAM register, the FSM change to the next state.

- In the state $\mathcal{S}_{13}$ is waiting to validate the $DONE\_S$ signal. It computes the last step of the algorithm.

Steps	Transition	Condition
Initialization	$\mathcal{S}_0$	reset = done
Steps (1) and (2)	$\mathcal{S}_0 \rightarrow \mathcal{S}_1$ $\mathcal{S}_1 \rightarrow \mathcal{S}_2$ $\mathcal{S}_2 \rightarrow \mathcal{S}_3$ $\mathcal{S}_3 \rightarrow \mathcal{S}_4$	$DONE\_C = 1$ $DONE\_C' = 1$
Step (3)	$\mathcal{S}_4 \rightarrow \mathcal{S}_5$ $\mathcal{S}_5 \rightarrow \mathcal{S}_6$ $\mathcal{S}_6 \rightarrow \mathcal{S}_7$ $\mathcal{S}_7 \rightarrow \mathcal{S}_8$	$DONE\_Q = 1$
Step (4)	$\mathcal{S}_8 \rightarrow \mathcal{S}_9$ $\mathcal{S}_9 \rightarrow \mathcal{S}_{10}$ $\mathcal{S}_{10} \rightarrow \mathcal{S}_{11}$ $\mathcal{S}_{11} \rightarrow \mathcal{S}_{12}$	$DONE\_R = 1$
Step (5)	$\mathcal{S}_{12} \rightarrow \mathcal{S}_{13}$	$DONE\_S = 1$

Table 3: State transition table of the FSM

4.2. Our semi−parallel architecture

We present in this section our semi-parallel architecture which aim is to execute the maximum of intermediate operations in parallel in order to minimize the number of cycles. As an example, we describe in Fig. 9 a scheduling of the steps of the algorithm 3, with $n = 4$ and $w = 36$.

It appears that it is not necessary to wait for the end of the computation of the last coefficient of C in step (1) before to start the computation of the step (2) (Alg.4). Indeed, at step 1, we compute in parallel the terms that have to be added in order to get c'_0. Thus, the computation of the coefficients C'_0 starts at step 2 (Fig. 9). It is possible to observe the same thing with step (2) and step (3) and with step (4) and step (5).

As a consequence, our semi-parallel architecture is composed of 4 modules as detailed in Fig. 10, Fig. 11, Fig. 12 and Fig. 13. Each of them, take in charge one or two steps of Alg. 3. These modules are:

- Fig. 10: The multiplication $a_i \times b_j$ is done through the cell $MULTIPLIER$. The cell $ADDER$ collects the products and computes give c'_i.

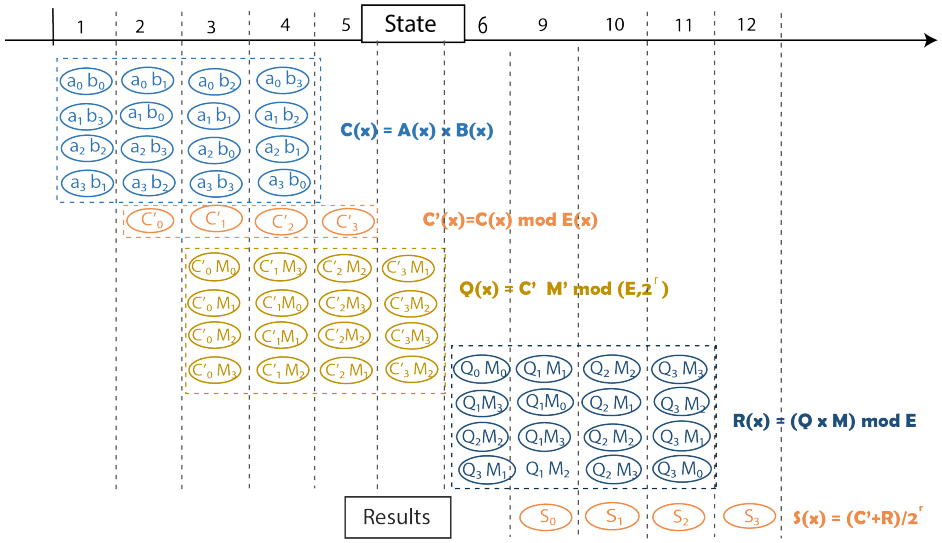

Figure 9: Scheduling of the semi-parallel architecture

- Fig. 11: The multiplication $c'_i \times m'_i \mod (\phi)$ generates the coefficients of polynomial Q.
- Fig. 12: The multiplication of the coefficients of Q by $M \mod E$ generate R.
- Fig. 13: The addition $C' + R$, followed by the exact division by $\mathbb{Z}^r$, generates the coefficients of the output S.

The first module in Fig. 10 is in charge of step (1) and (2). The first part of this module is composed of multipliers that compute the product of the 36-bit coefficients of A and B in one cycle. The resulting 72-bit coefficient is stored in the registers.

The second part of this module is composed of an Adder operator which collect the intermediate products and sum them in order to compute one coefficient of C. In this example: $C_i = C_{i0} + C_{i1} + C_{i2} + C_{i3}$. The output of the Adder is connected to a demultiplexer that output the coefficient as soon as its computation is completed. In this example, this module requires 5 cycles for completing the computation. It has to be noted that each state is executed in two clock cycles.

The module in Fig. 11 is designed to compute the step (3) of Alg. 3. It is composed of MAC operators that input a 39-bit coefficient of M' and a

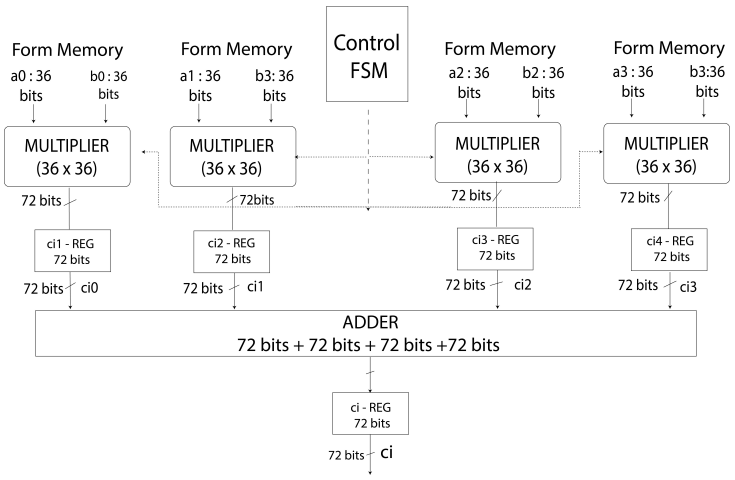

Figure 10: Semi-parallel architecture, computation of step (1) and (2) with $w = 36$ and a 128-bit modulus p

coefficient of C'. This coefficient is input serially. Each operator output a coefficient of Q that is written back serially to the memory.

The module operator in Fig. 12 computes the step (4). It is composed of multipliers that input the coefficients of M and Q. The multipliers are designed to perform the calculation of $(40 - bits \times 36 - bit$ integers) and output 76-bit integers. The result is stored in the $R_i - Reg$ registers that are input to the adder which computes one 76-bit coefficient of R. The output is connected to a demultiplexer which outputs all the computed coefficients of R.

The last module (Fig. 13) computes the last step of the modular multiplication in AMNS. It is composed of 76− Addershift operators that input the 72-bit coefficients of C' and the 76-bit coefficients of R. In this step a division by 2^r is performed. Thus, only the 36 most significant bits of the sum are output.

Main controller (MC). This design is controlled by a synchronous finite state machine (FSM). For $w = 36$ and a modulus p of 128 bits, the FSM controlling our semi-parallel architecture is illustrated in Fig. 14. It is composed of 11 states, the state $\mathcal{S}_0$ being the starting state. In this FSM, each state requires two cycles to be completed, except the final state which is executed in only one cycle. This leads to a total cost of 21 cycles. The data path of this

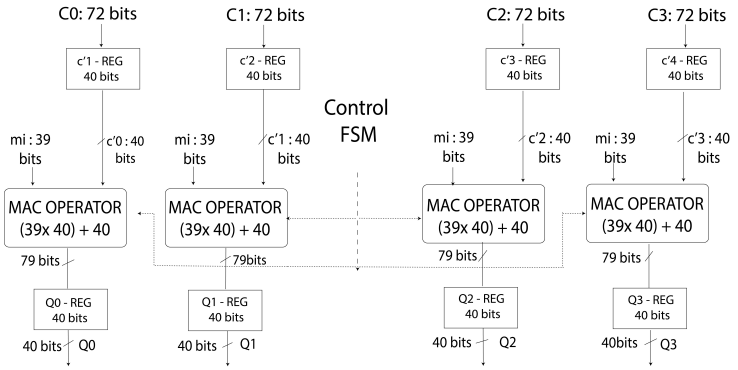

Figure 11: Semi-parallel architecture, computation of step (3) with $w = 36$ and a 128-bit modulus p

FSM consists of eight internal registers, a counter and a comparator. The controller remains in the state $\mathcal{S}_0$ until the START signal is set.

Once the START signal is set, the inputs a_i and b_i are loaded, the $C_{00}, C_{10}, C_{20}, C_{30}$ registers and the counter are reset.

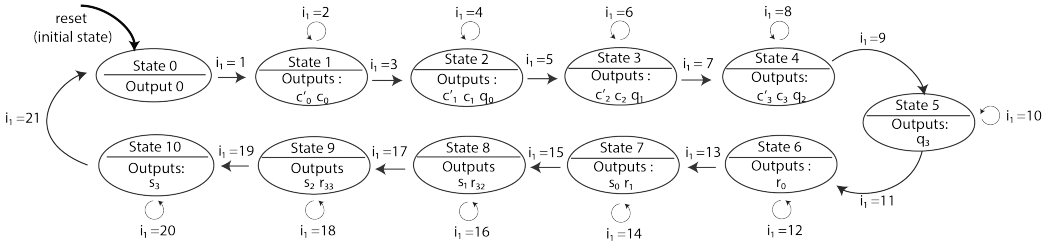

Figure 14: Finite state machine for the optimal architecture with $w = 36$ and a modulus p of 128 bits

The FSM loops while the DONE is not set. The machine advances to the next state once it is triggered by a valid transition.

- The first state is an initial state $\mathcal{S}_0$. During this state, the coefficients a_i and b_i are loaded into the registers.
- Once the states $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$ and $\mathcal{S}_4$ are completed, $DONE\_C'$ signal is set. These state permits the computations of the coefficients of the

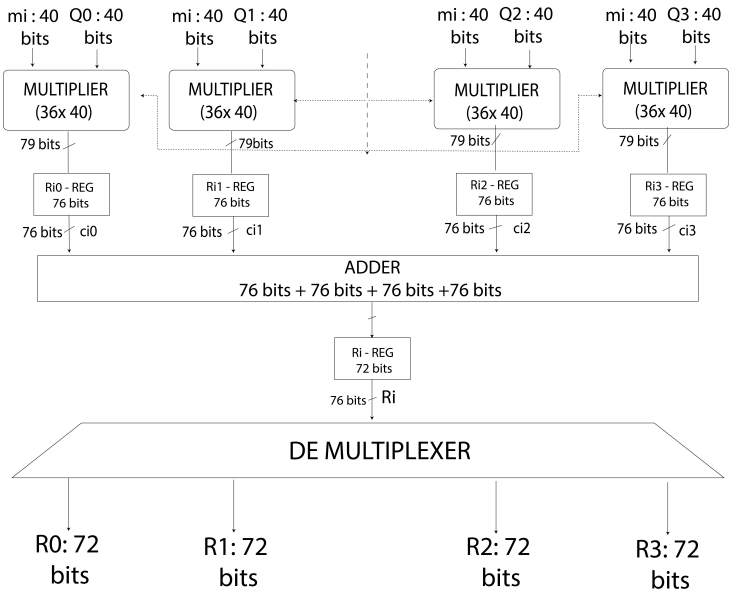

Figure 12: Semi-parallel architecture, computation of step (4) with $w = 36$ and a 128-bit modulus p

polynomial C' and charge them in the BRAM register. This corresponds to the steps (1) and (2)

- In the state $\mathcal{S}_1$ the FSM machine use an MAC operator to execute equation (1) and (2) of the algorithm 4. In this state the machine compute the coefficient c_0 of the polynomial C'.

- In the state $\mathcal{S}_2$ the machine compute the second coefficient c'_1 of C'. In parallel it starts the computation the coefficient q_0 of polynomial Q. Once the coefficients value of the polynomial q_0 and c'_1 are charged in the BRAM register, the FSM change to the next state.

- The state $\mathcal{S}_3$, $\mathcal{S}_4$ and $\mathcal{S}_5$ aim to validate the $DONE\_q_1$, $DONE\_q_2$, $DONE\_q_3$, $DONE\_c'_2$, $DONE\_c'_3$ signals. The FSM use eight MAC operators in each state to execute step (3). Once all the coefficients of the polynomial Q are generated, the FSM moves to the next state.

- The state $\mathcal{S}_6$ validate the $DONE\_r_0$ signal. It computes step (4),

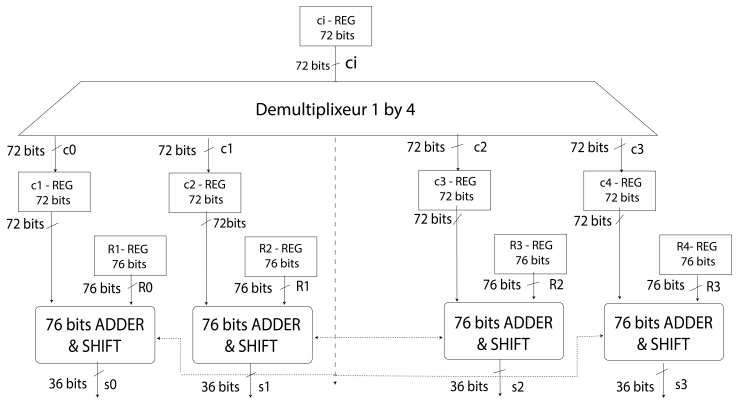

Figure 13: Semi-parallel architecture, computation of step (5) with $w = 36$ and a 128-bit modulus p

- The state $\mathcal{S}_7$, $\mathcal{S}_8$, $\mathcal{S}_9$ aim to validate the $DONE\_r_1$, $DONE\_r_2$, $DONE\_r_3$, $DONE\_s_0$, $DONE\_s_3$ and $DONE\_s_2$ signals. During these states, the coefficients of $Q \times M$ are computed in parallel. This corresponds to the step (5).

- The state $\mathcal{S}_{10}$ sets the $DONE\_s_3$ signal. It compute the fourth coefficient of the polynomial S by using an adder operator and a shift operator.

5. Experimental results

We completely implemented and validated our semi-parallel and the sequential architectures for AMNS on FPGA. We choosed two different families of FPGA, in order to confirm our results with two different technologies. Our designs have been described in VHDL language, synthetized, placed and routed with ISE 14.7 tools for Xilinx FPGA and Quartus II 12.1 for Intel-Altera FPGA using the default synthesis option The results are given for 4 FPGA families:

- two low cost FPGA: Aria II GX (EP2A6x45DF2915) and Zynq xc7 (z010-3g400),

- two high performance FPGA: Virtex 6 (X6vlx 75 t-3ff484) and Cyclone V (5cgXFC7DF 31 C8ES)

Below, we report the results for a modular multiplication with different coefficient sizes w and for different size of p: 128, 256, and 512 bits for FPGA that are large enough. Here we report the maximum operating frequency of the architectures, the number of cycles require to completed the modular multiplication, the total time of the computation, the number of DSP blcks and LUT needed in the implementation.

Family	Xilinx				
FPGA	Zynq		Virtex 6		
p size bit	128	256	128	256	512
w **bit**	36		36		72
Fq(Mhz)	256.5	242.3	238	225.8	191
Cycles	19	33	19	33	33
Time μs	0.074	0.136	0.079	0.146	0.172
DSP	60	56	60	120	236
LUT	1025	12105	1035	2738	18964

Table 4: Semi parallel architecture results on Xilinx FPGA

Family	Intel - Altera				
FPGA	Aria II		Cyclone V		
p size bit	128	256	128	256	512
w **bit**	36		36		72
Fq(Mhz)	218.2	194.4	157.1	106.4	101.6
Cycles	19	33	19	33	33
Time μs	0.087	0.169	0.12	0.309	0.324
DSP	70	128	40	80	112
LUT	1722	4173	1201	2747	14670

Table 5: Semi parallel architecture results on Intel-Altera FPGA

In Tab. 4 and 5 respectively, we give a report of the results of our semi-parallel architecture for Xilinx and Intel-Altera FPGA after placement and routing. The results of the implementations of our sequential architectures are reported in figure 6 and 7.

As expected, the semi-parallel architecture records the most efficient speed of execution while the sequential design achieves the smallest area.

We observe that Xilinx FPGA provides fastest computations for modular multiplication in AMNS representation compared to the Altera Family.

Family	Xilinx				
FPGA	Zynq		Virtex 6		
p size bit	128	256	128	256	512
w **bit**	36		36		72
Fq(Mhz)	283.2	255.7	309.5	189.7	224.1
Cycles	23	47	23	47	47
Time μs	0.081	0.183	0.074	0.247	0.209
DSP	51	53	51	91	216
LUT	791	2827	791	1740	24763

Table 6: Sequential architecture results on Xilinx FPGA

Family	Altera				
FPGA	Aria II		Cyclone V		
p size bit	128	256	128	256	512
w **bit**	36		36		72
Fq(Mhz)	237.3	213.3	157.2	123.4	82.7
Cycles	23	47	23	47	47
Time μs	0.096	0.22	0.146	0.38	0.568
DSP	42	80	28	52	112
LUT	1267	2423	679	1304	14283

Table 7: Sequential architecture results on Intel-Altera FPGA

6. Comparison with the state of the art

To our knowledge our implementation is the first VHDL implementation of a FPGA architecture of a modular multiplication using AMNS as a number system. We compared our design to some previously published works:

- The first design we are comparing to is detailed in Ors et al. (2003). It is a systolic architecture for Montgomery Modular multiplication and uses binary numbers.

- The second design is also a systolic architecture that is based on the CIOS Montgomery modular multiplication by Mrabet et al. (2016). It also make use of binary numbers.
- The third design is detailed in Bigou and Tisserand (2015). Its originality is that it uses *Residue Number Systems* (RNS) as an integer number system. This architecture is based on cox-rower components introduced in Kawamura et al. (2000).
- The design published in Rezai and Keshavarzi (2014) uses carry-save representation in the intermediate results.
- The systolic architecture described in Rezai and Keshavarzi (2016) uses the binary signed digits representation in order to minimize the number of non-zero partial product in the modular multiplication.
- Similarly the architecture presented in Rezai and Keshavarzi (2015) make use of a binary signed digits.

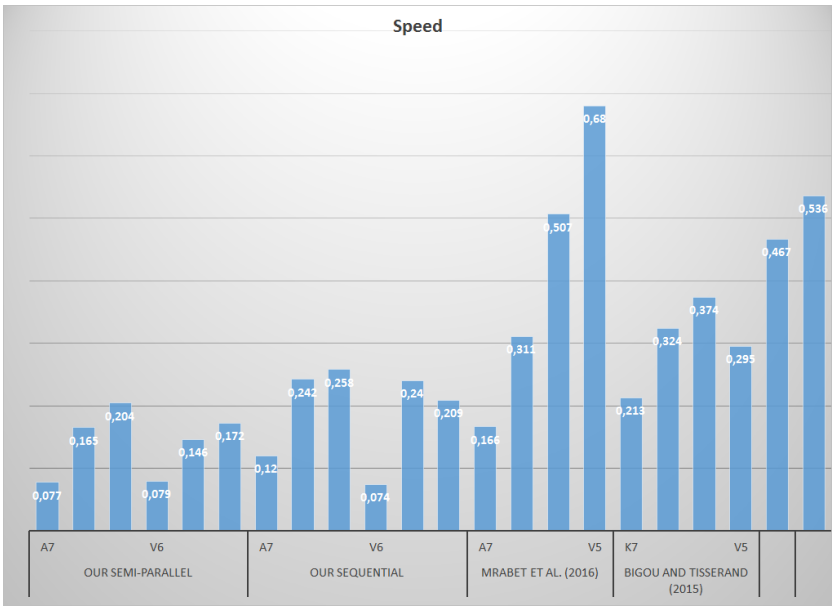

Figure 15: Speed test results

Comparing FPGA design is a challenging task because the technologies used are heterogeneous. In this comparison we detailed the number of slices,

the number of DSP blocks, the number of cycles needed to complete the modular multiplication and the total delay. Some architectures uses DSP blocks, while some others use only LUT blocks. For the sake of comparison, we unified the measure of the area by calculating an estimate value of $LUT_{eq} = nLUT + (nDSP \times r)$, where $nLUT$ is the number of LUT slices, $nDSP$ is the number of DSP blocks and r is the ratio of number of LUTs by the number of DSP blocks available on the target device. Thus the is Area-Delay Product (ADP) is computed as follows: $LUT_{eq} \times Time(\mu s)$.

Architecture	FPGA	p size	DSP	LUT	Cycle	Time (μs)	ADP
Semi-parallel	A7	128	60	1563	19	0.077	901.1
		256	120	2728	33	0.165	3796.3
		512	188	29985	33	0.204	12598.4
	V6	128	60	1035	19	0.079	844.9
		256	120	2738	33	0.146	3220.4
		512	236	18964	33	0.172	9797.1
Sequential	A7	128	51	790	23	0.12	1129
		256	91	1718	47	0.242	4137.4
		512	176	37138	47	0.258	17255.5
	V6	128	51	791	23	0.074	666.1
		256	91	1740	47	0.24	3933.8
		512	216	24763	47	0.209	12443.6
Mrabet et al. (2016)	A7	128	19	355	33	0.166	591.9
		256	33	809	33	0.311	1986
		512	87	2650	33	0.507	8797.9
Bigou and Tisserand (2015)	K7	192	18	999	58	0.213	864.5
		384	41	2111	58	0.324	2942.2
		512	56	8757	66	0.374	6835.5
	V5	192	15	1447	58	0.295	741
		384	42	2256	58	0.467	2446.1
		512	57	10877	66	0.536	7999.2
Ors et al. (2015)	V-E	128	-	806	388	1.807	1456.4
		256	-	1548	772	7.686	11897.9
		512	-	2972	1450	16.17	48057.2

Rezai and Keshavarzi (2014)	V5	512	-	3048	-	0.449	1400
Rezai and Keshavarzi (2015)	V5	512	-	6091	-	0.851	5200

Table 8: Comparison of our work with state of the art implementations for modulus p from 128 bits up to 512 bits.

We present in table 8 the implementation results of our architectures compared to the state of the art for modulus size between 128 and 512 bits. Such a modulus size range targets implementations for ECC (Hankerson and Menezes (2011)). We highlight the area, time and throughput results for two sizes of the modulus p. The table 9 show the results for 256-bits moduli. This size targets Elliptic Curve Cryptograhy. In table 10 we show the results for large cryptographic sizes that target RSA algorithm (Rivest et al. (1978a)). In this table the target moduli have 1024 bits.

Architecture	FPGA	LUTeq	Time (μs)	ADP	Throughput
Semi-parallel	A7	23008	0.165	3796.32	1544.3
	V6	22058	0.146	3220.468	1751.8
Sequential	A7	17097	0.242	4137.47	1055.5
	V6	16391	0.24	3933.84	1033.5
Mrabet et al. (2016)	A7	6386	0.311	1986.04	822.3
Bigou and Tisserand (2015)	K7	9081	0.324	2942.24	1185.1
	V5	5238	0.467	2446.146	820.9
Ors et al. (2003)	V-E	1548	7.686	11897.928	33.3

Table 9: Comparison of ADP and throughput result for 256-bit modulus p.

Our semi-parallel implementation achieves a higher throughput than the sequential. Although the ADP of semi-parallel improve significantly, they do not approach the minimum of ADP in all technologies. The performance of semi-parallel architecture is generally better than sequential architecture.

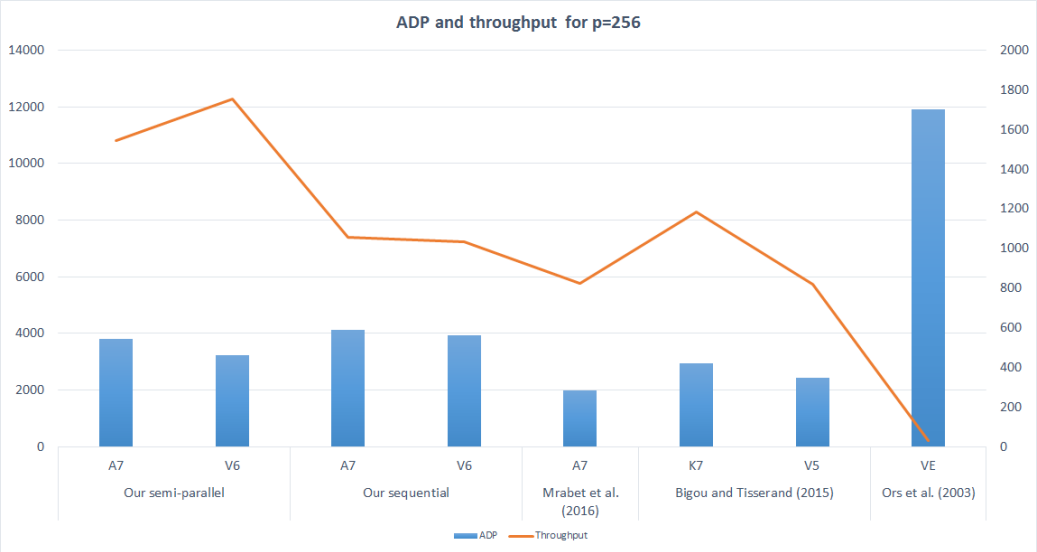

Figure 16: ADP and throughput results

It appears that our designs faster than the other designs for 256-bit moduli as well for 1024 moduli. As a consequence, the throughput is the highest for all moduli size. This is mainly due to the number of DSP blocks that greatly increase the speed of the architecture, at the price of the area. Compared to the design described in Ors et al. (2003), our architectures are up to 23 times faster and a ADP up to 1.6 times smaller.

Compared to the architecture of Mrabet et al. (2016) which is a systolic hardware architectures of Montgomery modular multiplication, our design is a bit faster with a better throughput and a bit higher.

The design of Bigou and Tisserand (2015) is an original design that make use of Residue Number Systems as number system. Compared to this architecture, our designs remain faster but larger. The throughput of our architecture is is higher.

Considering the comparison with the modular multiplication architectures described in (Rezai and Keshavarzi (2016)),)Rezai and Keshavarzi (2014)) and (Rezai and Keshavarzi (2015)), our sequential design is faster at the cost of a larger number of slices. We can see that the throughput achieved by these implementations is significantly smaller than what we are able to achieve with our proposed semi-parallel and sequential implementations.

Regardless to the size of the modulus p both our sequential and semi-

parallel designs have an interesting property: the number of cycle is independent from the size of the modulus.

Architecture	FPGA	Area (LUT_{eq})	Time (μs)	ADP	Throughput (Mb/s)
Semi-parallel	V7	124151	0.43	53384	2381.39
Sequential	V7	55667	0.598	33288	1712.37
Rezai and Keshavarzi (2014)	V5	6105	0.883	5390	1159.6
Rezai and Keshavarzi (2016)	V6	1125	2.56	2880	356
Rezai and Keshavarzi (2015)	V5	6091	0.851	5180	1203.3

Table 10: Comparison of ADP and throughput result for 1024-bit modulus p..

7. Conclusion and future works

In this paper, we present a fast hardware architecture that realizes a modular multiplication over large prime characteristic finite fields $\mathbb{F}_p$, which uses AMNS representations in order to speed up modular multiplication. We propose two hardware architectures: a semi-parallel architecture and a sequential one. Our semi-parallel architecture provides the fastest implementations. As we propose the first hardware implementation of the modular multiplication in AMNS, we compare our results with existing modular multiplication like systolic CIOS, systolic architectures using carry-save or signed digit representations for the intermediate results and arcitecture designed for the RNS modular multiplication. The comparison of our results with the state of the art highlight that we propose the fastest implementation of modular multiplication for large field application with the highest throughput.

For future work, we prospect to optimize the semi-parallel structure to reduce the area size. We would like to implement a full ECC scalar multiplication using the modular arithmetic in AMNS.

References

A. Menezes, P. C. Van Oorschot, S.A.V., 1997. Handbook of applied cryptography. CRC Press.

Bajard, J.C., Imbert, L., Plantard, T., 2004. Modular number systems: Beyond the mersenne family, in: Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, pp. 159–169.

Bajard, J.C., Imbert, L., Plantard, T., 2005. Arithmetic operations in the polynomial modular number system, in: 17th IEEE Symposium on Computer Arithmetic (ARITH-17) 2005, Cape Cod, MA, USA, pp. 206–213. Extended (complete) version available at: `https://hal-lirmm.ccsd.cnrs.fr/lirmm-00109201/document`.

Barrett, P., 1987. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor, in: Odlyzko, A.M. (Ed.), Advances in Cryptology — CRYPTO' 86, Springer, Berlin, Heidelberg. pp. 311–323.

Bigou, K., Tisserand, A., 2015. Single base modular multiplication for efficient hardware rns implementations of ecc, in: CHES: 17th International Workshop on Cryptographic Hardware and Embedded Systems, Springer. pp. 123–140.

Blakely, G.R., 1983. A computer algorithm for calculating the product ab modulo m. IEEE Transactions on Computers C-32, 497–500. doi:`10.1109/TC.1983.1676262`.

Chaouch, A., Dosso, Y.F., Didier, L.S., El Mrabet, N., Ouni, B., Bouallegue, B., 2019. Hardware optimization on fpga for the modular multiplication in the amns representation, in: International Conference on Risks and Security of Internet and Systems, Springer. pp. 113–127.

Didier, L.S., Dosso, F.Y., Véron, P., 2020. Efficient modular operations using the Adapted Modular Number System. Journal of Cryptographic Engineering , 1–23DOI: 10.1007/s13389-019-00221-7.

El Mrabet, N., Gama, N., 2012. Efficient multiplication over extension fields, in: WAIFI, Springer. pp. 136–151.

H. Alrimeih, D.R., 2014. Fast and flexible hardware support for ecc over multiple standard prime fields. Transactions on Very Large Scale Integration (VLSI) Systems, 12, 2661–2674.

Hankerson, D., Menezes, A., 2011. Elliptic curve cryptography. Springer.

Kawamura, S., Koike, M., Sano, F., Shimbo, A., 2000. Cox-rower architecture for fast parallel montgomery multiplication, in: International Conference on the Theory and Applications of Cryptographic Techniques, Springer. pp. 523–538.

Montgomery, P.L., 1985. Modular multiplication without trial division. Mathematics of Computation 44, 519–521.

Mrabet, A., Mrabet, N.E., Lashermes, R., Rigaud, J.B., Bouallegue, B., Mesnager, S., Machhout, M., 2016. A systolic hardware architectures of montgomery modular multiplication for public key cryptosystems. IACR Cryptology ePrint Archive 2016, 487.

Negre, C., Plantard, T., 2008. Efficient modular arithmetic in adapted modular number system using lagrange representation, in: Information Security and Privacy, 13th Australasian Conference, ACISP 2008, Wollongong, Australia, pp. 463–477.

Ors, S.B., Batina, L., Preneel, B., Vandewalle, J., 2003. Hardware implementation of a montgomery modular multiplier in a systolic array, in: Proceedings International Parallel and Distributed Processing Symposium, pp. 8 pp.–. doi:10.1109/IPDPS.2003.1213341.

Plantard, T., 2005. Arithmétique modulaire pour la cryptographie. Ph.D. thesis. Montpellier 2 University, France.

Rezai, A., Keshavarzi, P., 2014. High-throughput modular multiplication and exponentiation algorithms using multibit-scan–multibit-shift technique. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 23, 1710–1719.

Rezai, A., Keshavarzi, P., 2015. Compact sd: A new encoding algorithm and its application in multiplication. International journal of computer mathematics 94, 554–569.

Rezai, A., Keshavarzi, P., 2016. High-performance scalable architecture for modular multiplication using a new digit-serial computation. Microelectronics journal 55, 169–178.

Rivest, R., Shamir, A., Adleman, L., 1978a. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21, 120–126.

Rivest, R.L., Shamir, A., Adleman, L., 1978b. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21, 120–126.

Stein, W., al., 2018. Sagemath. `http://www.sagemath.org/index.html`.

Takagi, N., 1992. A radix-4 modular multiplication hardware algorithm for modular exponentiation. IEEE Transactions on Computers 41, 949–956. doi:`10.1109/12.156537`.

Taylor, F., 1981. Large moduli multipliers for signal processing. IEEE Transactions on circuits and systems 28, 731–736.