VHDL Implementation of modular arithmetic using Adapted Modular Number System representation

Louis Noyez, Mathieu Poignant

26 March 2021

Abstract

English : Modern cryptography protocols often rely on modular arithmetic to provide safe encryption. In the near future cryptography might change radically with the advent of the quantum computer. It is important to be able to accelerate these cryptography algorithms in order to guarantee safe communication can continue. Parallel representations of numbers are especially suited to this task. The Adapted Modular Representation System (AMNS) is one such system which represents numbers by polynomials, whose properties allow for faster arithmetic operations. This project has consisted in the development of a hardware implementation capable of handling AMNS operations using the hardware description language VHDL on a Zedboard development board target. We first learned about the theory behind AMNS before exploring different ways to implement operations. We used a development board to validate the function of our design and ultimately we compared the execution time of our system against other modular arithmetic algorithms and gave qualitative results about its performance.

Français : Les protocoles de cryptographie modernes dépendent souvent de l'arithmétique modulaire pour fournir un chiffrement sécurisé. Dans un futur proche la cryptographie pourrait changer radicalement avec le développement des ordinateurs quantiques. Il est important d'être capable d'accélérer ces algorithmes cryptographiques afin de garantir la perrenité des communications sécurisées. Les représentations "parallèles" des nombres sont particulièrement adaptées à cette tâche. L'Adapted Modular Numbering System (AMNS - Système de Numération Modulaire Adapté) est un de ces systèmes. Il représente les nombres par des polynomes, dont les propriétés permettent des opérations arithmétiques plus rapides. Ce projet a consisté au développement d'une implémentation matérielle capable de gérer l'arithmétique AMNS grâce au langage de description matériel VHDL vers une cible Zedboard (carte de développement). Nous avons d'abords découvert la théorie derrière l'AMNS avant d'explorer différentes méthodes d'implémentation des opérations. Nous avons utilisé une carte de développement pour valider le fonctionnement de notre design et enfin nous avons comparé le temps d'exécution de notre système à celui d'autres algorithmes d'arithmétique modulaire, et nous avons donné des résultats qualitatifs sur ses performances.

Contents

Introduction

Cryptography is used to encrypt almost all of the data transmitted in the world. Thus, there is a constant need for improvement of the efficiency of encryption methods, and optimization of the speed, power consumption and memory needs of implementations. Modern public-key cryptography algorithms often use modular arithmetic to provide encryption and make it difficult to decipher a message without the private key. Arithmetic operations are done on integers whose size ranges from hundreds of bits up to several thousand bits, which takes a lot of resources to implement and compute.

One of the paths towards optimization is to increase the speed at which arithmetic operations are done. For instance, representing a number with a large size by several independent numbers with smaller size and which can be handled simultaneously could be used to shorten computations. This is the principle behind the Adapted Modular Number System (AMNS) representation, in which large numbers are represented by polynomials, whose smaller coefficients are independent from one another which allows for parallel computations. Moreover, it has safety advantages that can offer countermeasures against some side channel attacks.

Since common processors are not optimized to perform operations in such representations, we have to implement these operations on hardware. Field Programmable Gate Array (FPGA) offers a good compromise between efficiency and ease of implementation.

The goal of this report is to present our hardware implementation of AMNS

arithmetic on the Zynq-7000 System on Chip using a Zedboard development board from Avnet, and to compare its efficiency with other methods.

1 AMNS number representation

1.1 Polynomial Modular Number System

A Polynomial Modular Number System (PMNS) **B** is defined by a tuple **(p, n, γ, ρ, E)** such that :

- p is a prime number.
- for any integer $0 \leq x < p$, there exists a polynomial **T** in $\mathbb{Z}_{n-1}[X]$ such that $\deg(T) \leq n-1$ and $x = T(\gamma) \pmod p = \sum_{i=0}^{n-1} t_i \gamma^i \pmod p$, with $|t_i| < \rho$ and α and λ small integers. T is called representative of x.
- E is a polynomial with $E(X) = X^n - \alpha X - \lambda$ and $E(\gamma) = 0 \pmod p$.

This polynomial representation is very useful to run parallel calculations. For instance every coefficient of the result of a polynomial addition can be computed simultaneously.

The result of some operations can leave the domain of PMNS :

- let $n \geq 2$, let U and $V \in$ B such that $\deg(U) = \deg(V) = n-1$
  $\deg(U * V) = 2n-2$ and $U * V = \sum_{i=0}^{2n-2} r_i X^i$, with $|r_i| \geq \rho$.
  $\deg(U * V) > n-1$ and $|r_i| \geq \rho \implies U * V \notin$ B.

To bring back the result in the PMNS domain, its degree is first reduced using an External Reduction operation which consists in the polynomial operation $\mod(E(X))$. Its coefficients are then bound by ρ through an Internal Reduction operation using the RedCoeff algorithm described further.

1.2 Adapted Modular Number System

An Adapted Modular Number System (AMNS) is a variation of a PMNS where $E(X) = X^n - \lambda$. This shape of E for allows for a very efficient External Reduction :

- let U and $V \in$ B, $W = U * V = W_1 + X^n * W_2$
  $\deg(W_1) \leq n-1$ and $\deg(W_1) \leq n-2$.

$$\begin{aligned} W &= W_1 + X^n * W_2 \\ &= W_1 + (X^n - \lambda + \lambda)W_2 \\ &= E * W_2 + (W_1 + \lambda W_2) \end{aligned}$$

$$W' = U * V \mod (E) = W_1 + \lambda W_2$$

$$E(\gamma) \equiv 0 \mod p \implies W(\gamma) \equiv W'(\gamma) \equiv U(\gamma) * V(\gamma) \mod p$$

$$\deg(W') \leq n-1$$

This external reduction algorithm can be implemented by tweaking a regular polynomial multiplication, with just a few additional λ-multiplications. W' now has the right degree, but its coefficients might still be too large.

1.2.1 Internal reduction

This algorithm is similar to the Montgomery modular reduction method and needs additional parameters : an integer ϕ and two polynomials from the AMNS domain

Algorithm 3 RedCoeff (Coefficient reduction), [26]

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

Ensure: $S(\gamma) = V(\gamma)\phi^{-1} \pmod{p}$

1: $Q \leftarrow V \times M' \bmod (E, \phi)$
2: $T \leftarrow Q \times M \bmod E$
3: $S \leftarrow (V + T)/\phi$
4: return S

Figure 1: RedCoeff Algorithm for Internal Reduction

M and M' such that $M(\gamma) = 0 \pmod{p}$ and $M' = -M^{-1} \bmod (E, \phi)$.

The operation $\bmod(E, \phi)$ corresponds to a modulo E followed by a modulo ϕ on the coefficients of the resulting polynomial. The modulo ϕ can be efficiently implemented if ϕ is chosen power of 2. Often, ϕ is chosen to be the same size as a machine word to make the modulo ϕ even easier.

Because this algorithm return S such that $S(\gamma) = V(\gamma) * \phi^{-1} \pmod{p}$, we need to convert the numbers into a Montgomery domain. It consists in choosing Va representation of $v * \phi$ instead of v. Thus, RedCoeff$(U * V)$ is a representation of $u * v * \phi^2 * \phi^{-1} = u * v * \phi$ which is the representation of $u * v$ in the Montgomery domain.

However, that is not the case after an addition because RedCoeff$(U + V)$ is a representation of $(u + v) * \phi * \phi^{-1} = u + v$ which is not in the Montgomery domain. Thus, before performing RedCoeff on $U + V$, we have to multiply the result of the addition by the representative of 1 in the Montgomery domain. Then, after computing RedCoeff, we get the representative of $(u + v) * 1 * \phi^2 * \phi^{-1} = (u + v) * \phi$

Algorithm 6 Conversion from classical representation to AMNS

Require: $a \in \mathbb{Z}/p\mathbb{Z}$ and $\mathcal{B} = (p, n, \gamma, \rho, E)$

Ensure: $A \equiv (a\phi)_{\mathcal{B}}$

1: $t = (t_{n-1}, ..., t_0)_\rho$ # radix-ρ decomposition of a

2: $U \leftarrow \sum_{i=0}^{n-1} t_i P_i(X)$

3: $A \leftarrow \text{RedCoeff}(U)$

4: return A

Figure 2: Conversion Algorithm from Integer to AMNS representation

which is in the Montgomery domain.

1.2.2 AMNS conversions

Because the numbers need to be in the Montgomery domain of the AMNS to perform RedCoeff, we use the conversion algorithm presented in [1] to determine the representations of integers :

This algorithm needs the radix-ρ decomposition of the integer and the $P_i(X)$ which are the representatives of $\rho^i\phi^2$ in B. Though, the radix-ρ is easy to implement if ρ is a power of 2 and the $P_i(X)$ can be precomputed.

Then, after doing the operations needed in the AMNS, we have to convert the numbers from the Montgomery domain of AMNS to get an integer. To do that, we use the conversion algorithm presented in [1] :

The algorithm uses RedCoeff to go out of the Montgomery domain by multiplying by ϕ^{-1} and, because $g_i = \gamma^i$, a corresponds to $A(\gamma)\phi^{-1} \pmod p$.

Algorithm 9 Conversion from AMNS to classical representation

Require: $A \in \mathbb{Z}_n[X]$, $\mathcal{B} = (p, n, \gamma, \rho, E)$ and $g_i \equiv \gamma^i \pmod{p}$, for $i = 1, \ldots, n-1$

Ensure: $a = A(\gamma)\phi^{-1} \pmod{p}$

1: $A \leftarrow \text{RedCoeff}(A)$
2: $a \leftarrow a_0$
3: **for** $i = 1 \ldots n-1$ **do**
4: $\quad a \leftarrow a + a_i g_i$
5: **end for**
6: $a \leftarrow a \pmod{p}$
7: return a

Figure 3: Conversion Algorithm from AMNS to Integer representation

Since the g_i can also be precomputed, the algorithm does not require any particular structure.

1.2.3 Maximum number of consecutive additions

It is proved in [1] (Proposition 2) that if $\rho \geq 2n|\lambda| \left\|M\right\|_\infty$ and $\phi \geq 2n|\lambda|\rho(\delta+1)^2$, and if U and V are such that $\left\|U\right\|_\infty$ and $\left\|V\right\|_\infty < (\delta+1)\rho$, then RedCoeff($U * V \bmod(E)) \in$ B.

Thus, because each polynomial $U \in$ B has $\left\|U\right\|_\infty < \rho$, it is possible to add $\delta + 1$ elements of B before performing the modular multiplication followed by RedCoeff, and the result will still be in B.

δ can be determined with the parameters of the AMNS or it can be chosen to correspond to the needs of the application, thus placing a constraint on the choice of the parameters.

Algorithm 10 ExactRedCoeff

Require: $V \in \mathbb{Z}_n[X]$, $P_0 \equiv (\phi^2)_{\mathcal{B}}$ and $\mathcal{B} = (p, n, \gamma, \rho, E)$

Ensure: $S(\gamma) \equiv V(\gamma) \pmod{p}$

1: $T \leftarrow \text{RedCoeff}(V)$

2: $U \leftarrow T \times P_0 \bmod E$

3: $S \leftarrow \text{RedCoeff}(U)$

4: return S

Figure 4: ExactRedCoeff Algorithm for ensuring results stay within the Montgomery Domain

If more than $\delta + 1$ additions have been executed and we need to bring back the result in the AMNS, then we can apply the algorithm presented in [1] :

This algorithm uses P_0 representation of ϕ^2 in the AMNS and calls twice the algorithm RedCoeff. Thus, it is quite expensive in time and power consumption to compute.

Because there ares two calls of RedCoeff, if V is a representative of $v\phi$, S is the representative in B of $v * \phi * \phi^{-1} * \phi^2 * \phi^{-1} = v * \phi$.

2 VHDL Implementation

The objective of our project was the implementation of the Internal Reduction operation. This operation uses the Montgomery-like algorithm RedCoeff, which involves one polynomial multiplication mod (E, ϕ), one polynomial multiplication mod (E),a polynomial addition, and a division by Φ.

Algorithm 3 RedCoeff (Coefficient reduction), [26]

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

Ensure: $S(\gamma) = V(\gamma)\phi^{-1} \pmod{p}$

1: $Q \leftarrow V \times M' \bmod (E, \phi)$
2: $T \leftarrow Q \times M \bmod E$
3: $S \leftarrow (V + T)/\phi$
4: return S

In this algorithm, the polynomial multiplication mod (E) constitutes an arithmetic building block, thus it was the first operation we sought to implement.

2.1 Modular multiplication implementation

Let $A, B \in B_{AMNS}$ such that $A = \sum_{i=0}^{n-1} a_i X^i$,$B = \sum_{i=0}^{n-1} b_i X^i$, $R = A * B = \sum_{i=0}^{2n-2} r_i X^i$.

Let $0 \leq k \leq n-1$, $r_k X^k = \sum_{\substack{i+j=k \\ i,j \in [\![0\ ;\ n-1]\!]}} a_i b_j X^k$

Let n $\leq k \leq 2n-2, r_k X^k = \sum_{\substack{i+j=k \\ n \leq i+j \leq 2n-2}} a_i b_j X^k = X^n \sum_{\substack{i+j=(k-n)+n \\ n \leq i+j \leq 2n-2}} a_i b_j X^{k-n}$

Therefore $R^{'} = R \mod (E) = \sum_{k=0}^{n-1} r_k^{'} X^k$, with $r_k^{'} = \sum_{\substack{i+j=k \\ i,j \in [\![0\ ;\ n-1]\!]}} a_i b_j + \lambda \sum_{\substack{i+j=n+k \\ i,j \in [\![0\ ;\ n-1]\!]}} a_i b_j$

Once we determined a general formula to compute the result coefficients, we designed different calculation systems. We took inspiration and based our thought process around a C software implementation of an AMNS[1].

2.1.1 Flat implementation

Our first naive implementation consisted in designing with maximum parallel computing in mind. Below is an examples for $n = 4$:

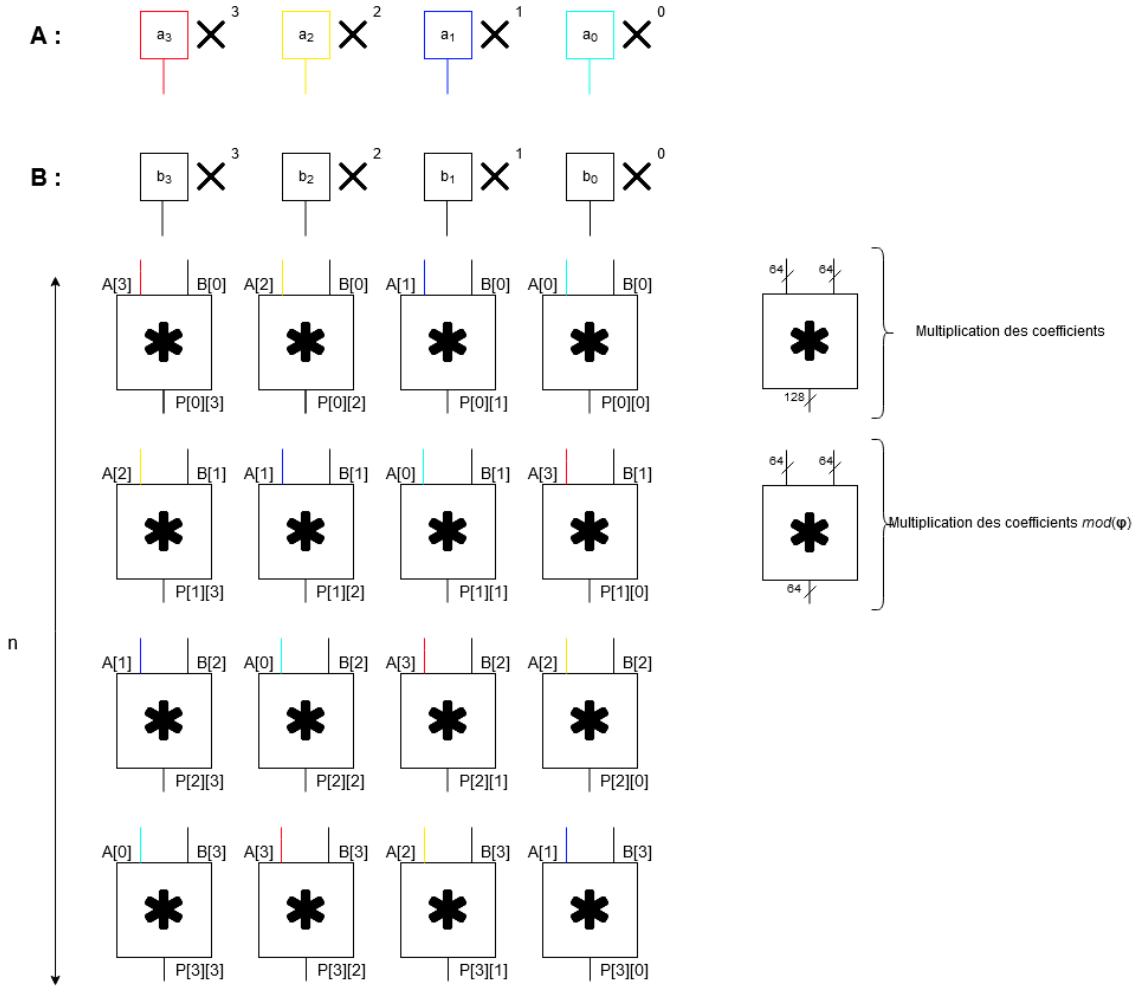


Figure 5: Flat implementation multiplier stage

This first multiplicative stage computes every pairwise products simultaneously.

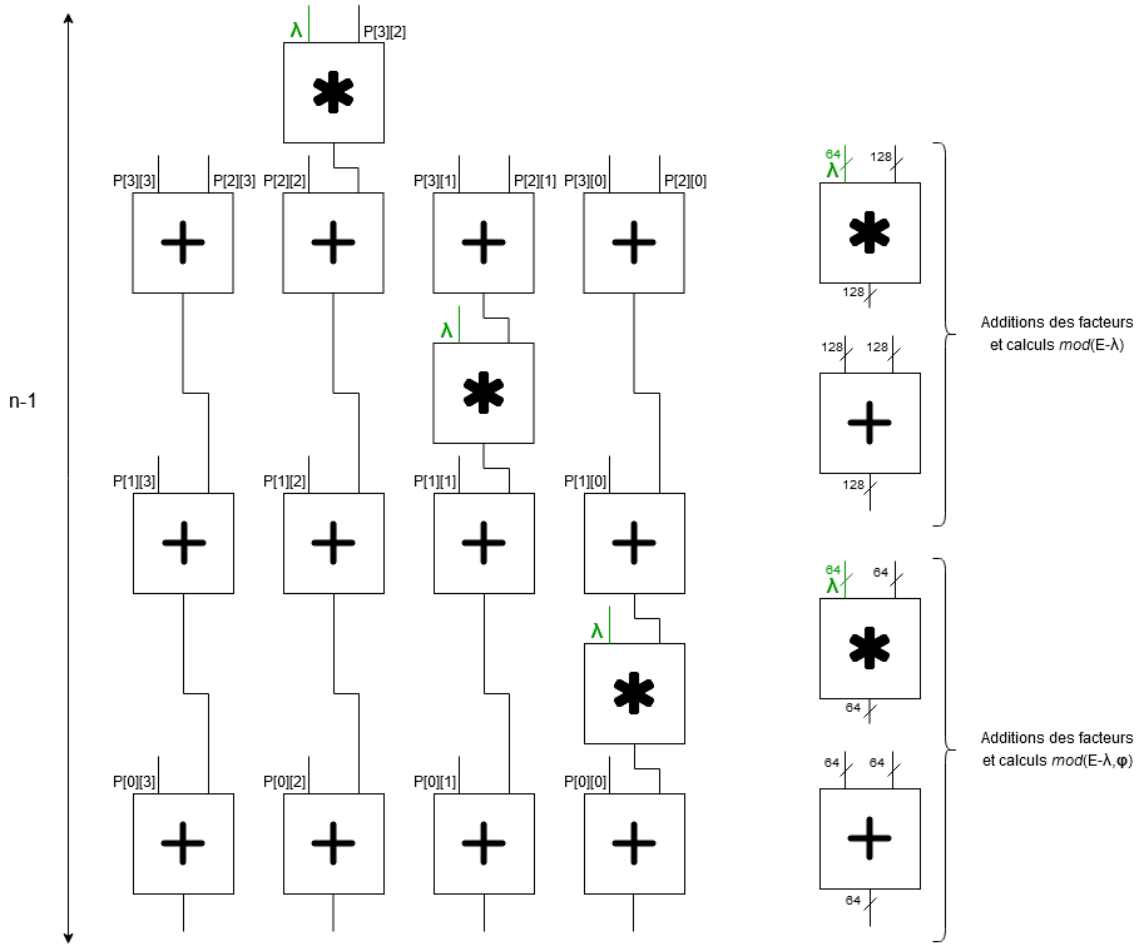


Figure 6: Flat implementation adder stage

This second stage computes the additions and multiplications by λ required to calculate the output coefficients. A pattern is clearly visible. This fully combinatorial architecture computes the result in one clock cycle, however it is actually extremely inefficient because the propagation time is very high, which implies a small frequency of use and high power consumption. Furthermore, this implementation would use a ludicrous amount of logical resources. When synthesized, all the Digital Signal

Processors (DSP) available on the Zedboard are used.

2.1.2 Folded implementation

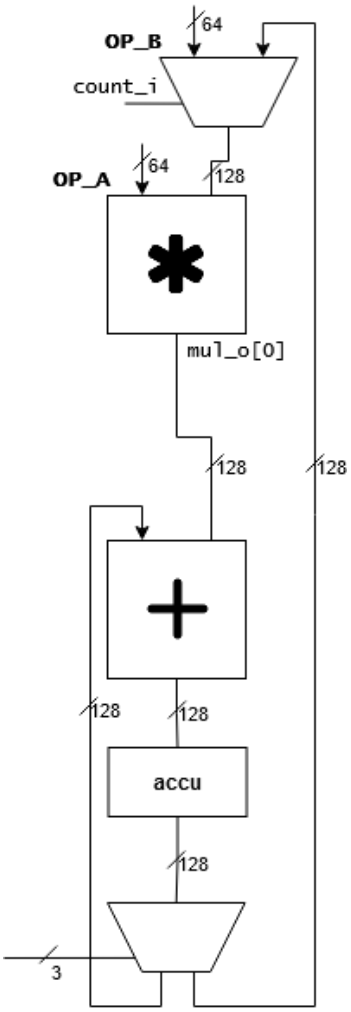


Figure 7: Single Arithmetic element used to compute a polynomial multiplication mod (E)

Our second attempt was to try and find the smallest arithmetic unit we would need to implement the polynomial multiplication mod (E). We realised that it

requires nothing more than a series of multiplications, additions and accumulations. Thus we designed a base arithmetic stage capable of these operations. A multiplier is used to compute the pairwise coefficients, with are accumulated in a register through an adder. The register output can loop to the multiplier input in order to compute a λ-multiplication It takes much less logical resources than the previous implementation and has a higher operating frequency but computing a result takes n^2 clock cycles. It does not take advantage of the parallel nature of the AMNS representation either.

2.1.3 Polyvalent implementation

The final implementation we settled upon consists essentially of n parallel iterations of the previous implementation, each responsible for the computation of one result coefficient. This architecture takes full advantage of the parallel nature of AMNS and is a compromise between the two previous implementations, but incurs no additional memory cost (n accumulator registers used). It can compute a polynomial multiplication $\mod (E)$ in $n + 1$ clock cycles.

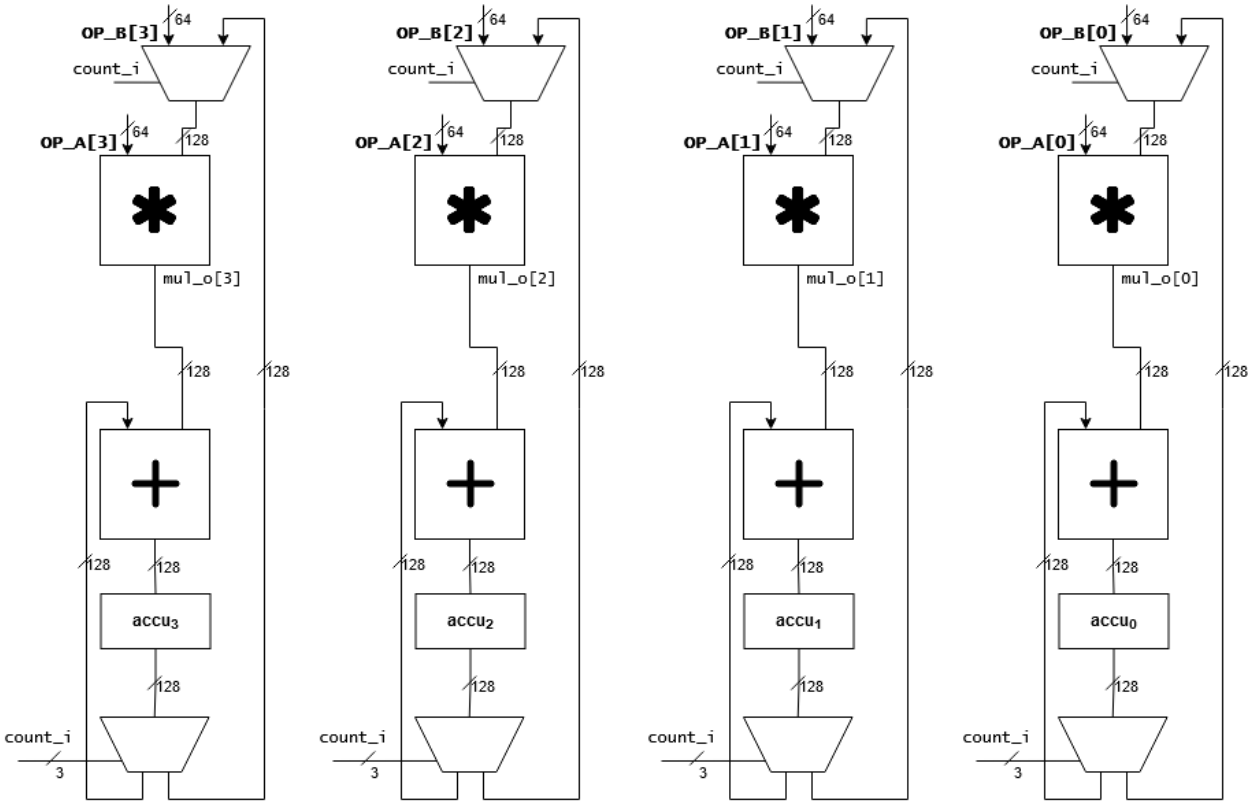


Figure 8: Arithmetic Unit including n base arithmetic bricks to promote parallel computation

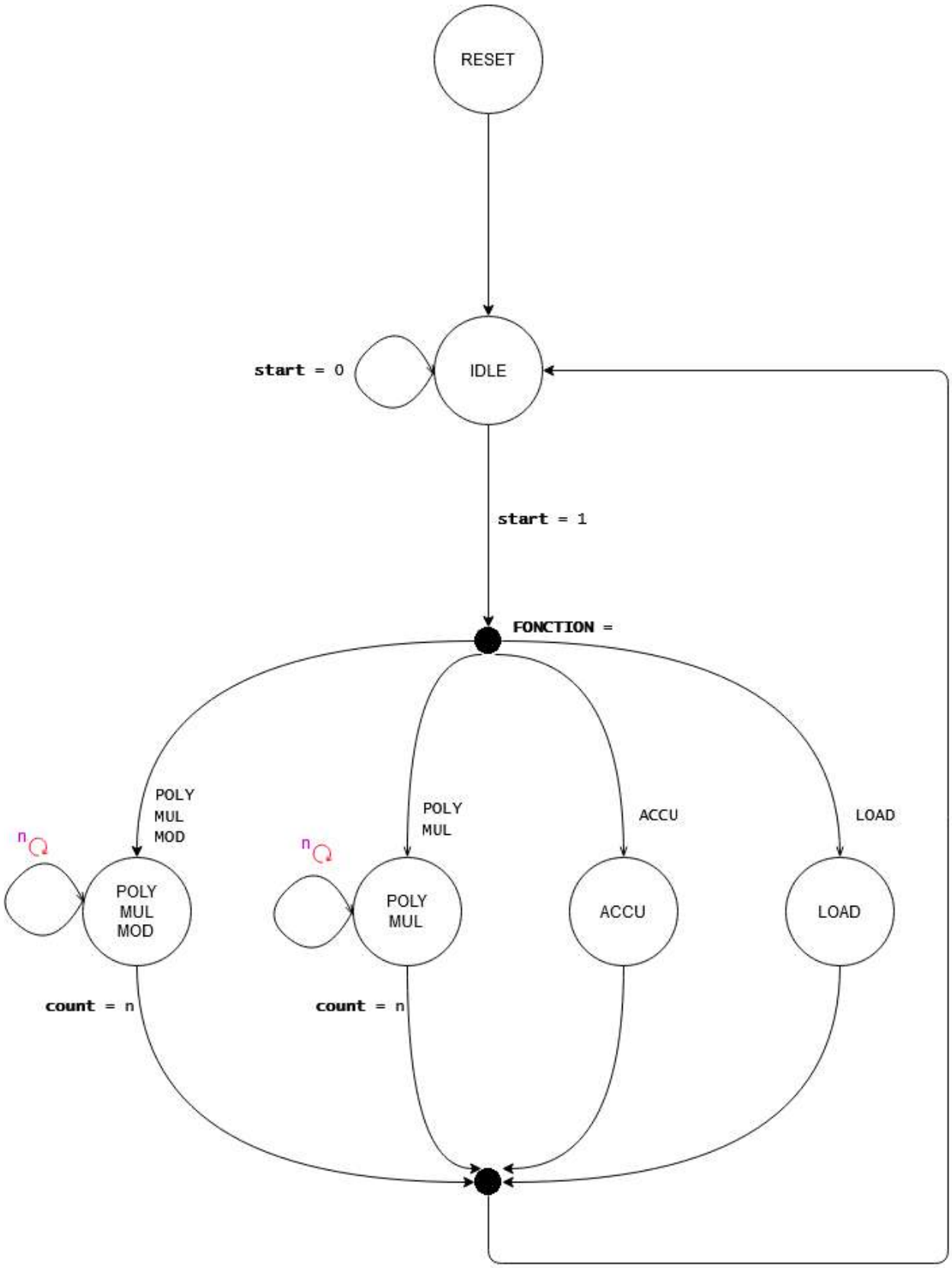
This structure also allows to perform a scalar multiplication if the accumulator has been reset beforehand, or it can perform the polynomial addition by redirecting the input data directly to the adder. Appendix (a) shows the FSM used to control the circuit and (b) shows the structure when performing the addition.

2.2 Control Systems - Finite State Machine

The arithmetic unit we have designed can be tweaked to implement other operations.

- The Polynomial multiplication mod (E) has been implemented
- The Polynomial multiplication mod (E, ϕ) and division by ϕ can be performed by selecting the right output bits since ϕ is a power of 2.
- The Polynomial addition of large coefficients can be performed by directly mapping a large input to the adder and accumulating the result.

These tweaks can be controlled by a Finite State Machine through a signal which changes the data path of the system.


RESET
IDLE
start = 0
start = 1
FONCTION =
POLY
MUL
MOD
POLY
MUL
ACCU
LOAD
n
n
POLY
MUL
MOD
POLY
MUL
ACCU
LOAD
count = n
count = n

Basic FSM

:

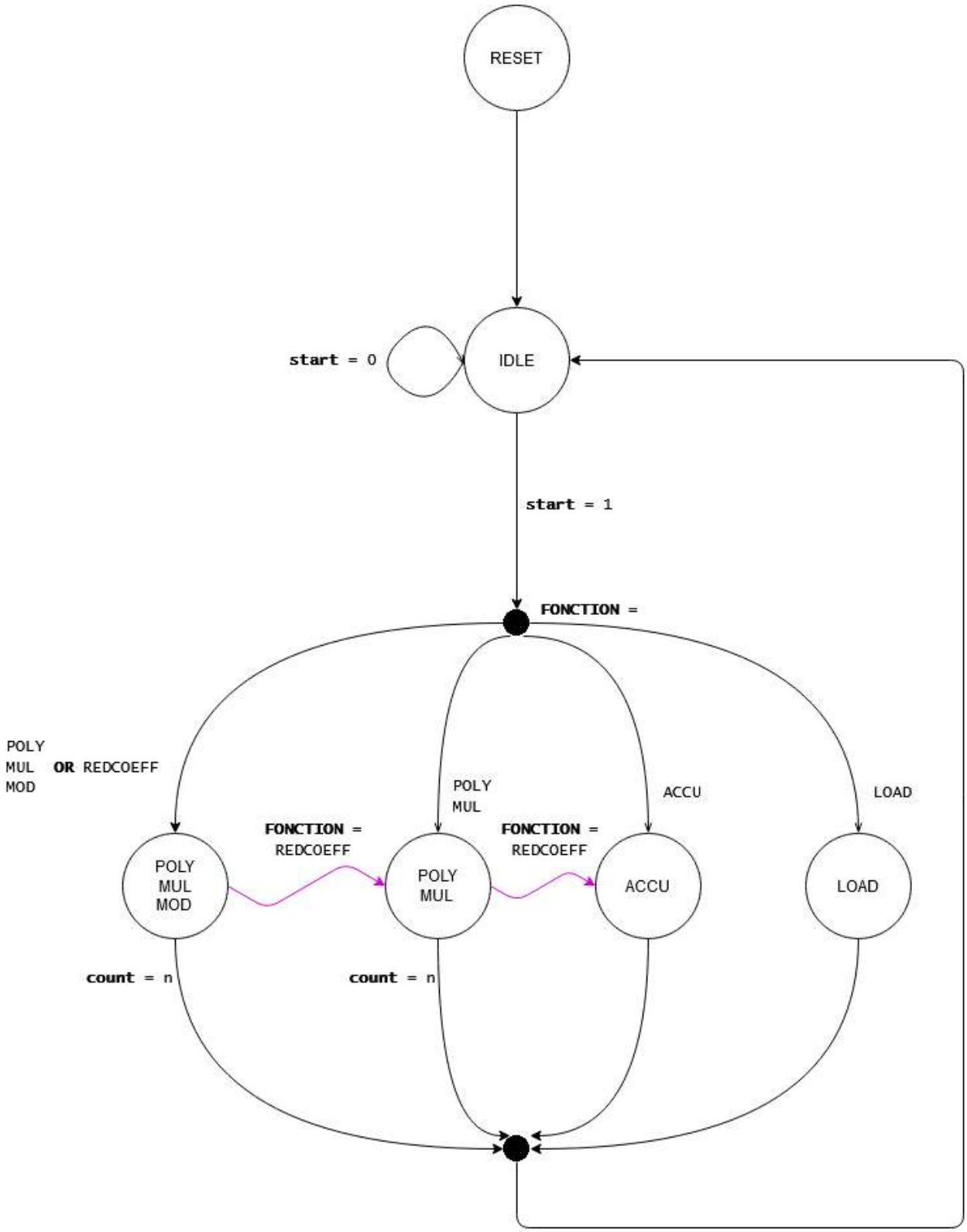
This architecture offers a simple FSM whose main states each correspond to a different operation. A function signal is used by an outside user to inform the FSM of which calculation is requested. A counter is also used to time execution of operations.

2.3 RedCoeff

Implementing RedCoeff only requires that we chain up the different operations we have described before, by adding a new value for our function signal, we can directly link different states without having to add new ones.

While signal $FONCTION_i$ has value "REDCOEFF", combinatorial logic also ensures that the right operands are chosen for the calculations and handles some operations such as divisions by ϕ.

This is the maximum level of operation automation our implementation provides. Indeed, to compute a multiplication in AMNS for instance, a user would first have to instruct our system to compute a polynomial multiplication, then an internal reduction independently. This is possible through the use of Zedboard development board. It includes a FPGA as well as an ARM-processor, which allowed us to write basic bare metal programs to communicate with our digital circuit.



Complete FSM

:

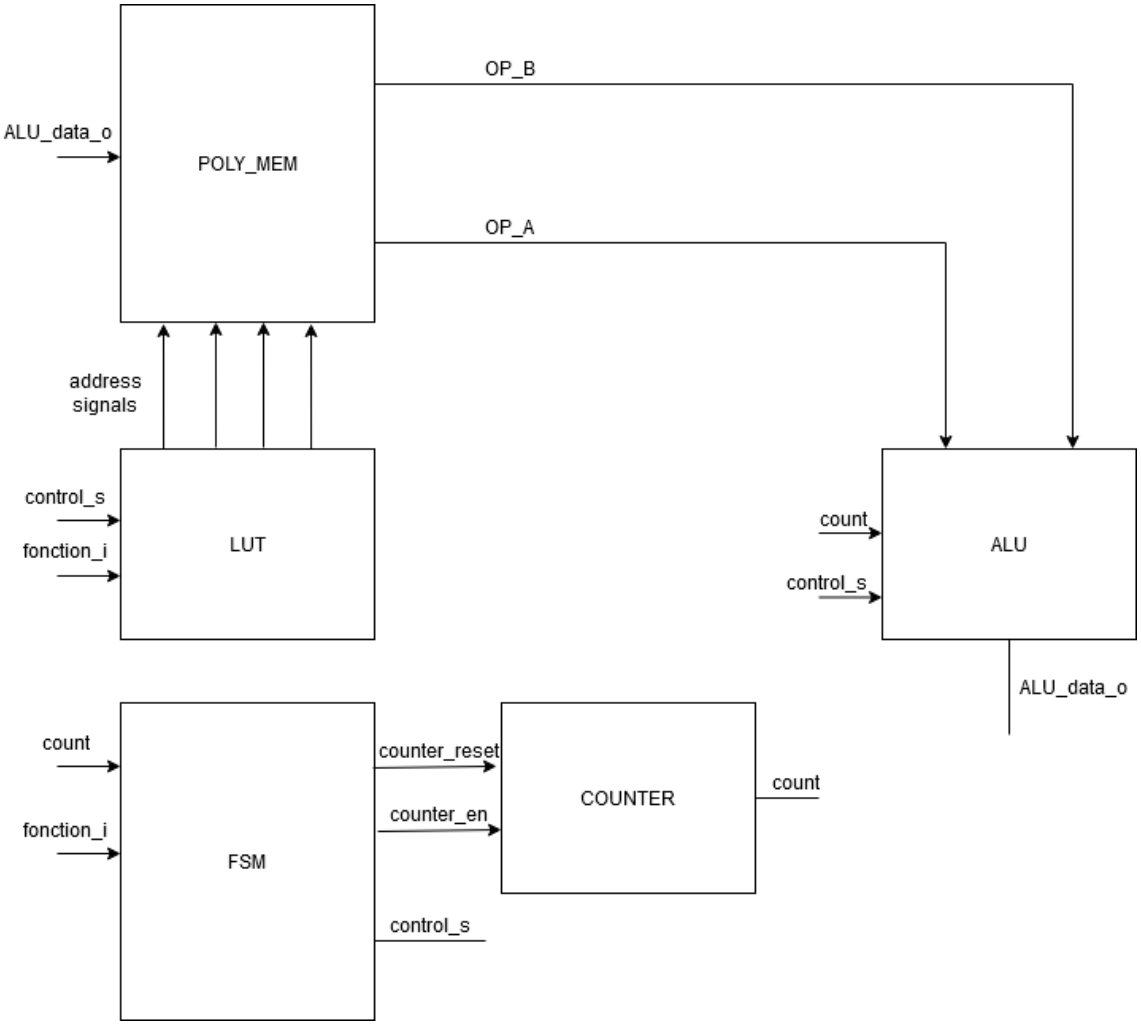


Figure 9: Block Diagram of the main entities the system is comprised of

Above is an illustration of the global structure of the hardware implementation. Besides the FSM, the counter and the ALU, which we have already mentioned, we've had to develop our own memory unit to ensure efficient selection of operands through use of a Lookup table and the signal fonction_i.

3 Results

3.1 Functional Verification

In order to validate the functionality of our design we had to implement verification steps using VHDL testbenches as well as computer algebra tools such as magma calculator, so as to exempt ourselves from the problematic of big numbers, and to obtain reliable data against which to compare our results. We were ultimately able to test our implementation on a development board Zedboard using Xillinx development tools.

3.2 Logical Resources

Our three different implementations all had different logical resources needs which might be suitable for different applications. However we observed that the relation "shortening of clock cycle/increase of logical resources has diminishing returns, the n Arithmetic elements ALU thus seems optimal given our implementation.

Implementation	DSP use (%)
Flat	100
Folded	10.3
Polyvalent	43.1

We can note that the flat implementation actually uses all of the available DSP which might indicate that it would also use non specialised entities to synthesize some arithmetic operations, which brings a definite loss of performance.

3.3 Timing Data

In order to compare the performance of our system to a more traditional implementation, we programmed the algorithm of the Montgomery Multiplication (an algorithm that can be used to accelerate modular multiplications). We then ported this algorithm onto the ARM processor available through the ZedBoard. 100 random test vectors where submitted to both our hardware system and the Montgomery multiplication system. By timing these operations, we were ultimately able to compare the average speed of execution of each method.

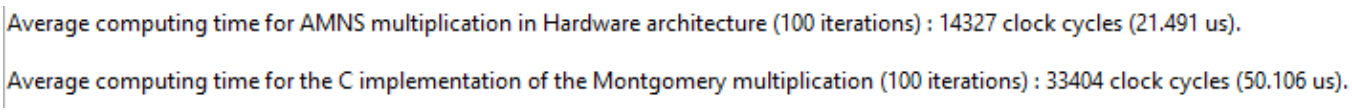


Figure 10: Average execution time of both systems for 100 random test-vectors (Communication with Zedboard through UART)

Implementation	time (clock)	time (μs)
MgtMul	33404	50.106
AMNS_Mul	14327	21.491

In the same environment, the hardware we have designed is on average two times faster than a competing method.

Conclusion

Although the design we have produced can be optimized significantly (through methods such as pipe-lining, ...), we find it has successfully taken advantage of the parallel nature of AMNS and that it highlights a few of the compromises that must be made when designing for parallel computing. We have also had the opportunity to note that theoretical work might still be done to optimize AMNS use, for instance by finding reduction polynomials with sparse representations.

References

[1] Laurent-Stéphane Didier, Fangan-Yssouf Dosso, and Pascal Véron. *Efficient modular operations using the adapted modular number system.* Journal of Cryptographic Engineering, Springer, 2020, 10.1007/s13389-019-00221-7. hal-02486345

Appendices

a) Illustration de la multiplication polynomiale $\mod(E)$ pour $n = 4$

$$
\begin{aligned}
&(a_3X^3 + a_2X^2 + a_1X^1 + a_0X^0) * (b_3X^3 + b_2X^2 + b_1X^1 + b_0X^0) \quad MOD(X^4 - \lambda) \\
=\ & a_3b_3X^6 + (a_3b_2 + a_2b_3)X^5 + (a_3b_1 + a_2b_2 + a_1b_3)X^4 \\
+\ & (a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3)X^3 + (a_2b_0 + a_1b_1 + a_0b_2)X^2 + (a_1b_0 + a_0b_1)X^1 + a_0b_0X^0 \quad MOD(X^4 - \lambda) \\
=\ & X^4(a_3b_3X^2 + (a_3b_2 + a_2b_3)X^1 + (a_3b_1 + a_2b_2 + a_1b_3)X^0) \\
+\ & (a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3)X^3 + (a_2b_0 + a_1b_1 + a_0b_2)X^2 + (a_1b_0 + a_0b_1)X^1 + a_0b_0X^0 \quad MOD(X^4 - \lambda) \\
=\ & (X^4 - \lambda + \lambda)(a_3b_3X^2 + (a_3b_2 + a_2b_3)X^1 + (a_3b_1 + a_2b_2 + a_1b_3)X^0) \\
+\ & (a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3)X^3 + (a_2b_0 + a_1b_1 + a_0b_2)X^2 + (a_1b_0 + a_0b_1)X^1 + a_0b_0X^0 \quad MOD(X^4 - \lambda) \\
=\ & (a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3)X^3 + (\lambda a_3b_3 + a_2b_0 + a_1b_1 + a_0b_2)X^2 + (\lambda(a_3b_2 + a_2b_3) + a_1b_0 + a_0b_1)X^1 + (\lambda(a_3b_1 + a_2b_2 + a_1b_3) + a_0b_1)X^0
\end{aligned}
$$

Figure 11: multiplication polynomiale $\mod(E)$

b)Polyvalent implementation performing addition