Master's degree thesis in Computer Engineering

High-speed polynomial multiplier to accelerate the arithmetical operations of Post-Quantum Cryptography algorithms

Supervisor:
Prof. Massimo Poncino
Dott. Emanuele Valea
Prof. Jahier Daniele Pagliari

Candidate:
Antonio Ras

A mia zia Ada,
che rimarrà per sempre
nel mio cuore

Contents

List of Figures

List of Tables

CHAPTER 1

Abstract

The advent of quantum computers and their increasing computing performance threatens the use of current cryptographic protocols as a way to ensure protection against cyberthreats. For this reason, in 2016, the American National Institute of Standards and Technology started a post-quantum cryptography standardization process for finding new quantum-resistant cryptographic protocols for both key encapsulation mechanisms and digital signatures. Saber is one of the four finalists, it relies on the Module-Learning-with-Rounding problem which is a lattice-based problem and it is believed to be quantum-resistant. The main implementation bottleneck of this protocol is the significant time spent in computing polynomial multiplications in polynomial rings with power of two moduli. This work aims at implementing a hardware architecture that can manage all the arithmetic operations contained in key generation, encryption and decryption functions of the Saber public key encryption protocol, for each of its versions. This is achieved using a schoolbook-based polynomial multiplication accelerator with different optimizations, that rely on centralized multiplication and the smallness of operand polynomials. Results from the design synthesis demonstrate good operating performance and low power dissipation values.

CHAPTER 2

Introduction

In the early 1980s, when Paul Benioff proposed a quantum mechanical model of the Turing machine [1], some scientists, including the theoretical physicist Richard Feynman, predicted that the computing power of quantum computers would lead to simulating things that a classic computer could not feasibly do [2]. Although there were some small realization of quantum computers with computing capabilities in the late 1990s, many researchers considered the feasibility of a powerful quantum computer a distant dream [3] until, in recent years, research investments [4][5] on these computers have had exponential growth.

In October 2019, Google AI, in partnership with U.S. National Aeronautics and Space Administration (NASA), realized a 54-qubit quantum processor able to perform a complete task, using quantum computation, in 200 seconds that would be the equivalent to 10,000 years of computational time using a classical supercomputer [6].

A notable application for quantum computers is the attack of cryptographic systems, thus breaking the security of the current public-key cryptographic protocols, such as the RSA and elliptic curve Diffie-Hellman algorithms. In particular, the RSA algorithm relies on the difficulty, for classical computers, of factorizing the product of two prime numbers used to generate the keys in the cryptographic protocol, because it would require too much time.

By comparison, a powerful quantum computer could efficiently solve this problem using Shor's algorithm. It was developed in 1994 by Peter Shor and it could be used for solving integer factorization problems, resulting in the capability of decrypting RSA-encrypted communications [9]. Fortunately, scientists estimate that a quantum computer powerful enough to run this algorithm and break current cryptographic systems, could be feasible in the next 15 to 20 years [7].

Post-Quantum Cryptography [8] is a field of cryptography that is focused on designing quantum-resistant public-key primitives based on problems that are presumed to be computationally infeasible for both classic and quantum computers. In 2016, the American National Institute of Standards and Technology started a post-quantum cryptography standardization process for finding new quantum-resistant cryptographic proto-

cols for both key encapsulation mechanisms and digital signatures. At the end of the Round 3 of this challenge, the finalists are four: **Classic McEliece**[12], **CRYSTALS-KYBER**[14], **NTRU**[13] and **SABER**[11]. Most of them are based on hard problems from lattice theory, that are presumed to be computationally infeasible even for quantum computers. The Round 4 submission [26] of the NIST challenge started on July 2022, after the end of the work presented in this thesis.

The focus of this work is Saber, which is a Chosen-Ciphertext Attack (CCA) resistant key encapsulation mechanism based on module lattices. Saber is based on the Module-Learning-With-Rounding problem [15]. It works with data represented as polynomials belonging to a polynomial ring with power of two moduli. The main implementation bottleneck of this protocol is the polynomial multiplication.

In the state of the art there are several types of algorithms that have been used to implement polynomial multipliers for accelerating the Saber scheme.

Zhu et al. [21] proposed an energy-efficient configurable crypto-processor supporting Saber multi-security-level key encapsulation mechanism. The polynomial multiplier is based on the Karatsuba algorithm [17], using an 8-level hierarchical structure. They designed a hardware efficient Karatsuba scheduling strategy and an optimized pre/post-processing structure to reduce the area overhead of the scheduling strategy. As a last optimization, they proposed a task-rescheduling-based pipeline strategy and truncated multipliers to enable fine-grained processing.

Bermudo Mera et al. [19] conceived a Took-Cook based polynomial multiplication implementation [18], introducing two optimizations in the algorithm itself, such as the evaluation and the interpolation steps, achieving a significant speed-up compared to the SW only implementation of the algorithm.

Wang et al. [23] proposed a HW/SW codesign solution for improving the polynomial multiplication on Espressif Systems 32 (ESP32) embedded microprocessor with low software overhead. Karatsuba and Toom-Cook algorithms are used in this implementation by applying the Kronecker substitution.

Roy et al. [20] proposed an instruction set coprocessor architecture for implementing the module lattice-based post-quantum key encapsulation mechanism Saber. They devised a parallel polynomial multiplier architecture, schoolbook-based algorithm, by exploiting some optimizations to reduce the latency imposed by the algorithm itself and the area on the Xilinx UltraScale+ FPGA.

Imran et al. [24] investigated how lattice-based algorithms work when implemented in hardware, using a polynomial multiplier based on the schoolbook architecture. The experiment is done by assuming that the algorithms will be implemented in an application-specific integrated circuit (ASIC) using a 65nm technology.

Another algorithm used to implement polynomial multiplication requires the use of the Number Theoretic Transform (NTT), which is the integer version of the Fast Fourier Transform (FFT). This algorithm is mainly used for CRYSTAL-KYBER implementations, being based on polynomial rings defined over finite fields with prime moduli.

Saber was conceived with the idea of making modular reductions as simple as possible, this is why it was chosen to use finite fields with powers of two as integer moduli. By implementing Saber using NTT, it would add a much higher latency cost and dissipated power [22] than the implementations of the same protocol using the polynomial multiplication algorithms mentioned above.
For these reasons, this work focuses on [20] to implement a HW architecture capable of handling all the arithmetic operations contained inside the Saber public key encryption function, for each of its security levels. To do this, a polynomial multiplier is implemented based on the schoolbook method. In the following it is reported the structure of this these. In Chapter 3 is introduced the relevant mathematical background, including a summary of the Saber PKE protocol. Chapter 4 discusses the polynomial multiplier operation, its optimization techniques and the design decisions that lead this proposed high-speed architecture. Chapter 5 illustrates how the architecture works during the three operating modes provided by the PKE functions. Chapter 6 explains the design verification of the devised architecture. Chapter 7 presents the implementation results and compares them with state-of-the-art solutions. The final chapter includes concluding remarks.

CHAPTER 3

Preliminaries

In this chapter, the notation used in the definition of the public key encryption (PKE) functions is firstly introduced.
The second section contains tables with all the constant values used during the execution of the arithmetic operations. This is useful especially during the design phase because they help to evaluate the complexity of the component. It shows also a first approach of the three possible security levels of Saber by mentioning both their parameter and constant values.
The third and fourth section provide a general description of Saber and its PKE functions, some pictures are shown for two reasons:

- Describe the three general types of arithmetic operations to be performed along the three PKE functions
- Give a simple overview of the steps to be followed to well-perform the operations, by covering all Saber protocol versions

3.1 Notation

Let p and q be two powers of 2, i.e. $p = 2^{\epsilon_p}$ and $q = 2^{\epsilon_q}$. Let define $\mathbb{Z}_p$ the ring of integers modulo p and let z mod p (z | p) be the reduction of z in [0,p). A ring of polynomials, denoted as $R_p = \frac{\mathbb{Z}_p[x]}{x^N+1}$, could be seen as a set of polynomials of degree N, whose coefficients belong to $\mathbb{Z}_p$.
Let l be an integer number, two further notations can be introduced: $R_p^{l\text{x}1}$ and $R_p^{l\text{x}l}$ are the array, represented in bold (i.e. $\mathbf{b}$), and the matrix, represented in upper case bold (i.e. $\mathbf{A}$), containing respectively l and lxl polynomials, each of these in R_p. This l parameter is useful because it identifies the rank of the lattice problem to be solved. In the following section l parameter will be also defined as the Saber version and it is correlated with its security level.
Every notation seen so far could be introduced by using another interger modulo q, that is the other power of two cited at the beginning of this section. Let v be a polynomial

in R_p, a left shift operation of k position can be defined using the symbol $v \ll k$. This is a coefficient-wise operation, it means that it is applied to all polynomial coefficients in the same way. It is also possible to define the right shift operation of k position on v denoted as $v \gg k$.
Let v, u be two polynomials in R_p, it is possible to perform coefficient-wise addition operation, so each i-th polynomial coefficient of v and u are added together. Finally let **b** be a polynomial array in $R_p^{l\text{x}1}$, the transposed operation, denoted as b^T, is the operation having as result the same polynomial array belonging in $R_p^{1\text{x}l}$.

3.2 Parameters and Constants

The parameters for Saber are:

- N : it represents the degree of the polynomial ring $R_p = \frac{\mathbb{Z}_p[x]}{x^N+1}$ which is the number of polynomial coefficients belonging to the ring. In Saber, $N = 256$ for all its versions.

- l : it determines the Saber version. The product $N * l$ is used to underlying the dimension of the lattice problem involved in the chosen protocol. An increase in the value of l leads to an increase in the size of the lattice problem to be solved and therefore to greater security.
  The possible values of this parameter and the name of Saber versions are:

  - $l = 2$: **LightSaber**
  - $l = 3$: **Saber**
  - $l = 4$: **FireSaber**

- q, p, T : they represent the moduli of the ring of integers involved in the scheme of the protocol. For an easier integer module reduction, these values are chosen to be powers of 2, in particular $q = 2^{\epsilon_q}$, $p = 2^{\epsilon_p}$, $T = 2^{\epsilon_T}$, with $\epsilon_q > \epsilon_p > \epsilon_T$. A higher choice for parameters p and T will result in lower security.

Below, a table with all the parameters described so far for the different types of Saber versions is shown:

Parameters	**LightSaber**	**Saber**	**FireSaber**
l	2	3	4
ϵ_q	13	13	13
ϵ_p	10	10	10
ϵ_T	3	4	6
q	2^{13}	2^{13}	2^{13}
p	2^{10}	2^{10}	2^{10}
T	2^3	2^4	2^6

Table 3.1: Parameter and constant values in Saber

The functions that will be described in the following section requires also the usage of three polynomial constants, two are polynomials and the last one is an array of polynomials: their values are determined by a mathematical formula involving the parameters shown in the previous table. These constants are very important during rounding phase because they are used during the first of the two steps to perform the operation correctly. A detailed description of how rounding operation works it is explained in the next section. In the following a table summarizing the constants described before is shown:

Constants	**Formula**	**LightSaber**	**Saber**	**FireSaber**
$h_1 \in R_q$	$2^{\epsilon_q-\epsilon_p-1}$	4	4	4
$h_2 \in R_q$	$2^{\epsilon_p-2} - 2^{\epsilon_p-\epsilon_T-1} + 2^{\epsilon_q-\epsilon_p-1}$	196	228	252
$\mathbf{h} \in R_q^{l\text{x}1}$	$2^{\epsilon_q-\epsilon_p-1}$	4	4	4

Table 3.2: Polynomial constants in Saber

3.3 Saber

Saber is an Indistinguishable under Chosen-Ciphertext Attacks (IND-CCA) secure Key Encapsulation Mechanism (KEM) based on module lattices, whose security relies on the hardness of the Module Learning With Rounding (M-LWR) problem, which is presumed to be computationally infeasible for both classic and quantum computers.
KEM schemes (Key Generation, Encapsulation and Decapsulation) are internally based on Public Key Encryption primitives (Key Generation, Encryption, Decryption) that contain the basic functions for the proper functioning of the protocol.
Public-key cryptography, also called asymmetric cryptography, is a system based on

the use of key pairs. Each pair consists of a public key, accessible to all, and a private key, which must be kept only by the owner and must not be accessible to anyone else. The PKE key generation function is responsible for generating this key pair. The use of the other two functions is established in the communication between a transmitter T and a receiver R, who wish to exchange information, which will have a pair of keys each. When T intends to send a message to R, it will use the receiver's public key to calculate the ciphertext via the PKE encryption function. The receiver, to read in clear the message that was sent by T, will use its secret key by applying the PKE decryption function on the encrypted message received. The security of the system depends on the secrecy with which the own preserves its secret key. It is precisely on this part that the operations of generating public and private keys, message encryption and decryption of ciphertext are focused. These three functions will be described in detail in the next section, emphasizing the operations that this work intends to optimize.
Saber is flexible, efficient and simple, it has been designed with features that make its implementation powerful, especially from the point of view of modular reduction. The flexibility of the cryptographic protocol leads to the need of just one hardware core to implement its multiple security levels.
Saber works with data represented as polynomials belonging to rings of polynomials whose integer moduli are powers of two. Modular operations generally have greater complexity due to the implementation of algorithms, such as Montgomery or Barrett, which can perform modular reduction on the operation itself. In the Saber protocol this complexity can be almost neglected since the whole integer moduli are powers of two. This leads, in hardware implementations, to replace the algorithms mentioned before with the use of simple arithmetic circuits or even without any additional components being applied, as in the case of this work.
Furthermore the M-LWR problem brings an advantage in the rounding phase. In Saber, this part consists in just two straightforward steps: adding constant polynomial values and performing a left/right shift operation for each polynomial coefficient involved in the operation. This two parts depend on the type of security level used for performing the cryptographic protocol and the PKE function where rounding operation has to be applied.

3.4 Saber PKE functions

The Saber Public Key Encryption (PKE) scheme is composed of three functions;

- **Key Generation** : it aims at generating a couple of public key and a secret key useful for a communication public key encryption based. it consists of the generation of a public matrix of polynomials called $\mathbf{A} \in R_q^{lxl}$ and a secret array of polynomials $\mathbf{s} \in R_q^{lx1}$. An array of polynomials called $\mathbf{b}$ is computed by scaling and rounding the product between $\mathbf{A}$ and $\mathbf{s}$. The function combines $\mathbf{A}$ and $\mathbf{b}$ as

public key and **s** as secret key. More precisely, due to the large size of the matrix **A**, the seed that generated that matrix itself is returned as part of the public key.

Algorithm 1: Saber.PKE.KeyGen()

$seed_{\boldsymbol{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{A} = \mathbf{gen}(\mathrm{seed}_{\boldsymbol{A}}) \in R_q^{l \times l}$
$r = \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{s} = \beta_\mu(R_q^{l \times 1}; r)$
$\boldsymbol{b} = ((\boldsymbol{A}^T \boldsymbol{s} + \boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
return $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}), \boldsymbol{s})$

Figure 3.1: Key Generation PKE function

- **Encryption** : this function is used for encrypt a message. It receives the public key generated in the previous function and the message that it is intended to be encrypted. Matrix **A** is generated starting from its seed and an array b' is computed with a secret key called s' that is generated specifically for encryption. A message can be encrypted by adding a scaled array $v' = b^T s'$, a constant polynomial h_1 and the message shifted and scaled properly. The final ciphertext c_m will be computed by applied shifting operation. The function returns as wrapped ciphertext c the final ciphertext itself c_m and the vector b' useful during decryption phase.

Algorithm 2: Saber.PKE.Enc$(pk = (\boldsymbol{b}, seed_{\boldsymbol{A}}), m \in R_2; r)$

$\boldsymbol{A} = \mathbf{gen}(\mathrm{seed}_{\boldsymbol{A}}) \in R_q^{l \times l}$
if *r is not specified* **then**
  $r = \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{s'} = \beta_\mu(R_q^{l \times 1}; r)$
$\boldsymbol{b'} = ((\boldsymbol{A}\boldsymbol{s'} + \boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
$v' = \boldsymbol{b}^T(\boldsymbol{s'} \bmod p) \in R_p$
$c_m = (v' + h_1 - 2^{\epsilon_p - 1} m \bmod p) \gg (\epsilon_p - \epsilon_T) \in R_T$
return $c := (c_m, \boldsymbol{b'})$

Figure 3.2: Encryption PKE function

- **Decryption** : this function is used to retrieve a message, that has been previously encrypted, starting from a given ciphertext. It receives the wrapped ciphertext c generated during encryption and the secret key **s** generated during key generation function. The message can be decrypted by recovering an approximation of $v^{'}$. The new array is obtained as $v = b'^T s$ scaled properly. This array is added with constant polynomial h_2 and a shifted ciphertext c_m. The final message m' is computed after a further shifting operation.

Algorithm 3: Saber.PKE.Dec($\boldsymbol{s}, c = (c_m, \boldsymbol{b'})$)

1 $v = \boldsymbol{b'}^T(\boldsymbol{s} \bmod p) \in R_p$
2 $m' = ((v - 2^{\epsilon_p - \epsilon_T} c_m + h_2) \bmod p) \gg (\epsilon_p - 1) \in R_2$
3 **return** m'

Figure 3.3: Decryption PKE function

The three functions seen so far are composed of three different types of arithmetic operations that this work aims to accelerate and optimize. The picture below shows these operations underlined with different colors.

Algorithm 1: Saber.PKE.KeyGen()

1 $seed_{\boldsymbol{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
2 $\boldsymbol{A} = \mathbf{gen}(\mathrm{seed}_{\boldsymbol{A}}) \in R_q^{l \times l}$
3 $r = \mathcal{U}(\{0,1\}^{256})$
4 $\boldsymbol{s} = \beta_\mu(R_q^{l \times 1}; r)$
5 $\boldsymbol{b} = ((\boldsymbol{A}^T \boldsymbol{s} + \boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
6 **return** $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}), \boldsymbol{s})$

Algorithm 2: Saber.PKE.Enc($pk = (\boldsymbol{b}, seed_{\boldsymbol{A}}), m \in R_2; r$)

1 $\boldsymbol{A} = \mathbf{gen}(\mathrm{seed}_{\boldsymbol{A}}) \in R_q^{l \times l}$
2 **if** *r is not specified* **then**
3 $\quad r = \mathcal{U}(\{0,1\}^{256})$
4 $\boldsymbol{s'} = \beta_\mu(R_q^{l \times 1}; r)$
5 $\boldsymbol{b'} = ((\boldsymbol{A}\boldsymbol{s'} + \boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
6 $v' = \boldsymbol{b}^T(\boldsymbol{s'} \bmod p) \in R_p$
7 $c_m = (v' + h_1 - 2^{\epsilon_p - 1} m \bmod p) \gg (\epsilon_p - \epsilon_T) \in R_T$
8 **return** $c := (c_m, \boldsymbol{b'})$

Algorithm 3: Saber.PKE.Dec($\boldsymbol{s}, c = (c_m, \boldsymbol{b'})$)

1 $v = \boldsymbol{b'}^T(\boldsymbol{s} \bmod p) \in R_p$
2 $m' = ((v - 2^{\epsilon_p - \epsilon_T} c_m + h_2) \bmod p) \gg (\epsilon_p - 1) \in R_2$
3 **return** m'

Figure 3.4: Arithmetic operation in PKE functions

Red operations are matrix-array multiplications with rounding. Yellow lines indicate the simple array-array multiplication, while in blue are underlined the message/ciphertext handling operation. This section will describe each of these types of arithmetic operation in different subsections by providing schemes and formula for a right understanding.

3.4.1 Array-array multiplication

The array-array polynomial multiplication is defined as the sum of the product of each i-th couple of the two input polynomial arrays.
Given $\mathbf{b}$ and $\mathbf{s} \in R_p^{l\text{x}1}$, let compute $v = b^T s \in R_p$ as

$$v = \sum_{i=0}^{l-1} b_i s_i$$

where l is the Saber security level (see Section 3.2)

3.4.2 Matrix-array multiplication with rounding operation

The matrix-array polynomial multiplication is defined as a consecutive array-array polynomial multiplication where the i-th polynomial result is obtained by multiplying the i-th matrix row times the input array polynomial.
Given $\mathbf{A} \in R_q^{lxl}$ and $\mathbf{s} \in R_q^{l\text{x}1}$, let compute $\hat{b} = A^T \mathbf{s} \in R_q^{l\text{x}1}$ as

$$\hat{b} = \sum_{k=0}^{l-1} \sum_{i=0}^{l-1} a_{ki} s_i$$

where l is the Saber security level (see Section 3.2)
After the matrix-array polynomial multiplication, the result has to be added with $\mathbf{h}$ constant polynomial array with a final three position shifting operation on each polynomial coefficient. The described operation above is summed up in the following flow diagram: In this work, for easier data management, it was decided to perform the

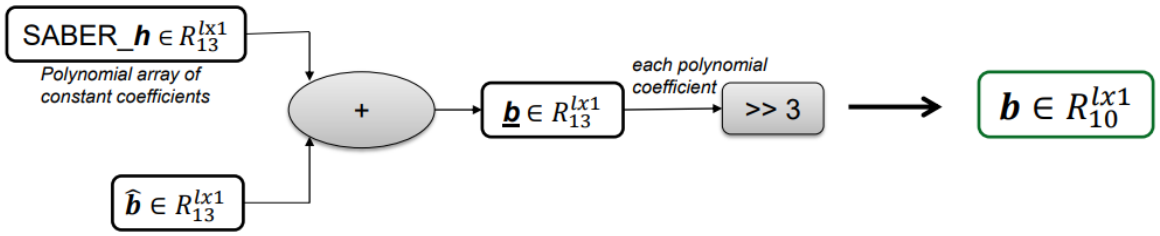

Figure 3.5: Rounding operation after matrix-array polynomial multiplication

rounding operation at the end of each array-array polynomial multiplication , using the constant SABER_h_1.

3.4.3 Message/ciphertext handling operation

The last two possible type of rounding that could be applied on arithmetic operations in PKE functions deal with message encryption and ciphertext decryption (the functions, described in Section 3.4 are shown again). Since these operations strictly depend on the level of security of the protocol, all possible cases for each version of Saber are schematized.

- **Message encryption** : The scheme below explains in detail all the steps to be taken in order to correctly execute line 7 of the PKE encryption function.

Algorithm 2: Saber.PKE.Enc$(pk = (\boldsymbol{b}, seed_{\boldsymbol{A}}), m \in R_2; r)$

1 $\boldsymbol{A} = \text{gen}(\text{seed}_{\boldsymbol{A}}) \in R_q^{l \times l}$
2 **if** *r is not specified* **then**
3 $\quad r = \mathcal{U}(\{0,1\}^{256})$
4 $\boldsymbol{s'} = \beta_\mu(R_q^{l \times 1}; r)$
5 $\boldsymbol{b'} = ((\boldsymbol{A}\boldsymbol{s'} + \boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
6 $v' = \boldsymbol{b}^T(\boldsymbol{s'} \bmod p) \in R_p$
7 $c_m = (v' + h_1 - 2^{\epsilon_p - 1} m \bmod p) \gg (\epsilon_p - \epsilon_T) \in R_T$
8 **return** $c := (c_m, \boldsymbol{b'})$

Figure 3.6: Encryption PKE function

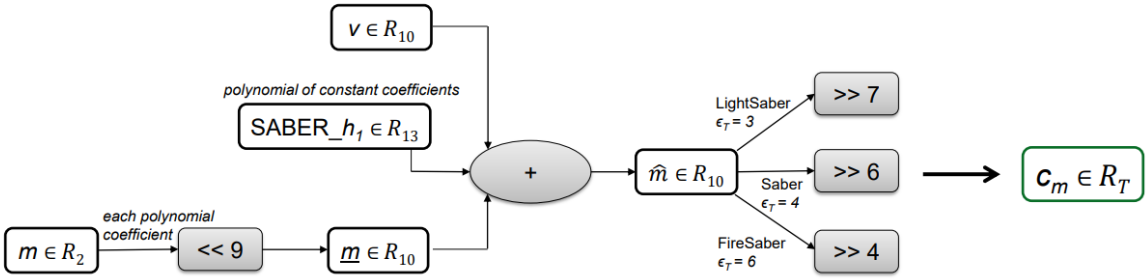

Figure 3.7: Operation flow for encrypting a message

Each polynomial coefficient of the message is firstly left shifted by nine position in order to be correctly added with constant polynomial h_1 and the v array previously computed by the function.
At this point, depending on the security level, each polynomial coefficient of the partial ciphertext will be right shifted by seven positions in case of LightSaber protocol, six positions for Saber and four positions for FireSaber.

- **Ciphertext decryption** : The scheme below explains in detail all the steps to be taken in order to correctly execute line 2 of the PKE decryption function.

Algorithm 3: Saber.PKE.Dec($\boldsymbol{s}, c = (c_m, \boldsymbol{b'})$)

1 $v = \boldsymbol{b}'^T (\boldsymbol{s} \bmod p) \in R_p$

2 $m' = ((v - 2^{\epsilon_p - \epsilon_T} c_m + h_2) \bmod p) \gg (\epsilon_p - 1) \in R_2$

3 **return** m'

Figure 3.8: Decryption PKE function

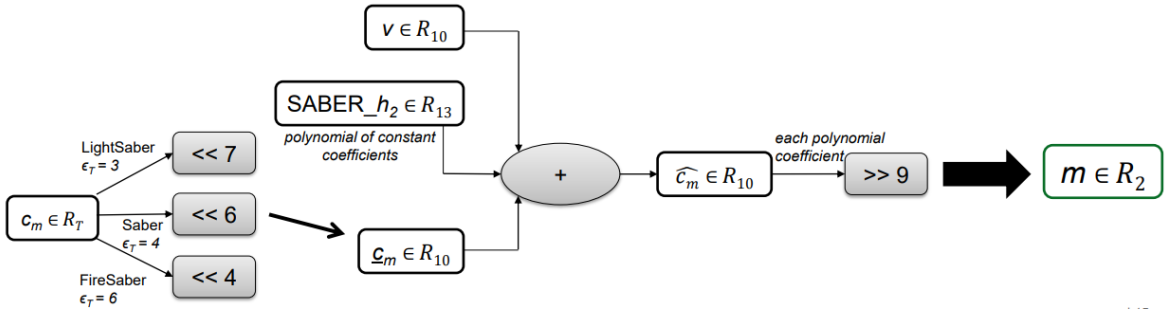

Figure 3.9: Operation flow for decrypting a ciphertext

Each polynomial coefficient of the ciphertext is firstly left shifted by a number of positions depending on the adopted security level. In particular, nine left positions for LightSaber, six positions four Saber and four positions for FireSaber. The result obtained after shift operation is added with the constant polynomial h_2 and the v array previously computed by the function.
At this point, each polynomial coefficient of the partial message obtained, will be right shifted by nine positions to compute the final message as output of the function.

CHAPTER 4

Architecture design

In this chapter it is firstly given an overview of the architecture design, its fundamental parts and its input/output ports with their possible values. In the next chapter, these values will be reported again for showing how they could be used.
The second section lists the internal organization of the datapath and provides a subsection for each sub-module, including their input/output signals. The most important part of this section is the polynomial multiplier. In order to explain how this multiplier works, the algorithm at the base of one single multiplication is described. As it is possible to see later, some technical optimizations are introduced and implemented in order to increase the design performance and to reduce the occupied area. Finally, this subsection explains the adopted modifications, inside the polynomial multiplier, for being able to correctly perform all the arithmetic operations of the algorithm described in Section 3.4.
Successively, it is shown how to solve the polynomial data storing problem and their synchronization by introducing the memory design and the decoder/buffers sizing respectively.
The last section deals with the control part. Specifically, the representation of the Finite State Machine (FSM) and the explanation of each of its states is shown.
For a better understanding, the FSM scheme is divided according to the colors representing the different arithmetic operations inside the PKE algorithms.

4.1 Overview

The architecture is split in two parts:

- **Datapath** : It contains all the functional units, such as the arithmetic logic unit, that perform data processing operations. In addition, there are memories with the task of ensuring the reading of the data used in the execution of the arithmetic operations and, finally, the writing of the results. Other sequential components, such as register buffers, are also inserted, having the task of synchronizing and speeding up the data flow within the datapath.
- **Finite State Machine** : It is the controller of the component, composed by a set of states. Each of them drives all the predefined signals to the datapath for issuing its execution. The FSM provides a set of transitions (the change from one state to another) that, step by step, make the result generation process successful.

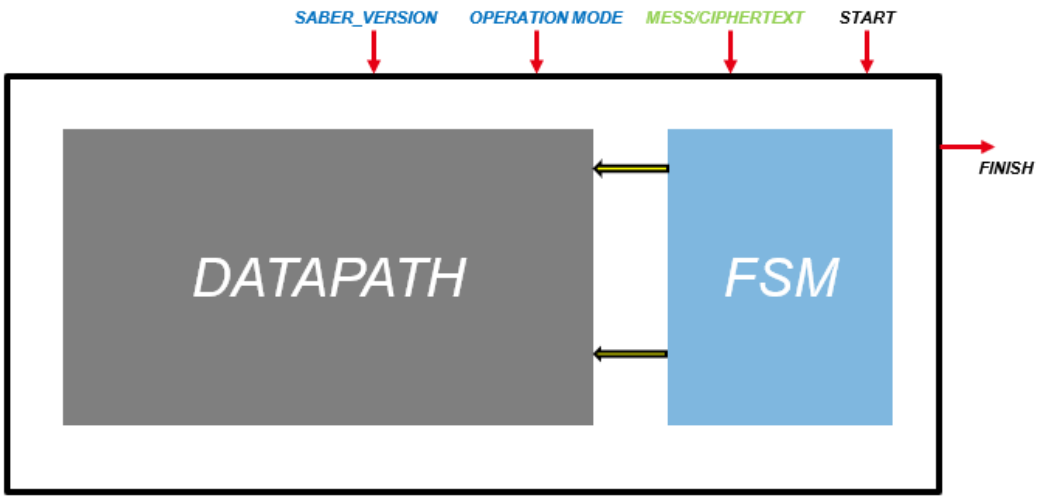

Figure 4.1: Architecture design model

The component has also some input/output ports.
In the following it is described, for each port, its purpose and its possible values:

- **saber_version** : it represents the Saber version (security level) that the component must take into account for its operation. All the possible values have been summarized in the table below:

Saber version input	Description
00	-
01	Light Saber
10	Saber
11	Fire Saber

Table 4.1: Description of the saber_version input values

- **operating_mode** : it represents the possible operation modes that the component can perform. All the possible values have been summarized in the table below:

Operation mode input	Description
00	Key Generation
01	Encryption
10	Decryption
11	-

Table 4.2: Description of the operating_mode input values

- **mess_ciphertext** : it contains the message to be encrypted during Encryption mode and the ciphertext to be decrypted during Decryption mode.

- **start** : it is used to start a given operation using a given Saber version. It is active high.

start input	Description
0	Idle
1	Begin operation

Table 4.3: Description of the start input values

- **finish** : it is the signal that will be raised whenever the component completes an operation.

finish output	Description
0	Operation not completed
1	Operation completed

Table 4.4: Description of the finish output values

4.2 Datapath

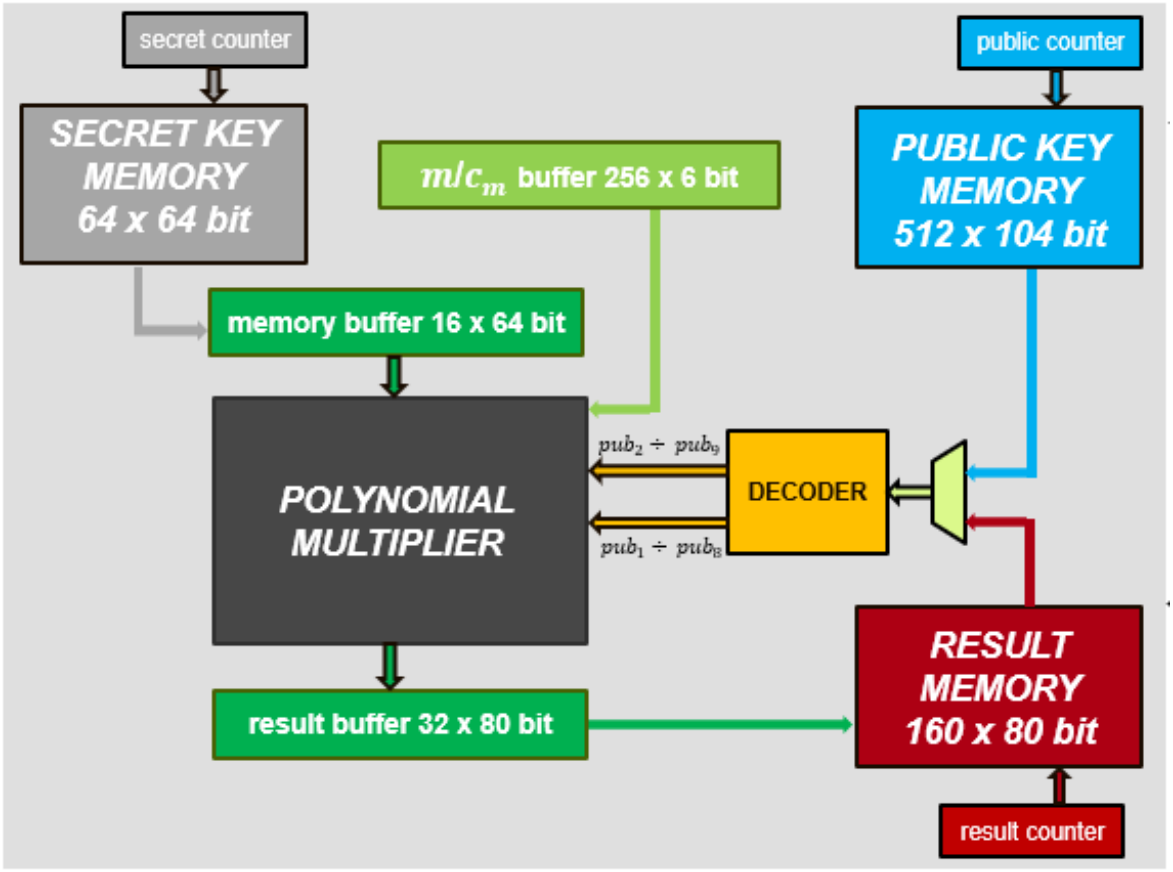

Figure 4.2: Datapath internal representation

The internal composition of the datapath is devised to ensure easy understanding of its operation. The memories in the upper part of Figure 4.2 have the task of saving the data to be processed, while the result key memory to save the results once the operation is finished. The memory buffer is instead positioned to store the next secret polynomial to be processed at the same time the multiplier is completing the multiplication using the previous one.
The result buffer is used to store the result at the end of a given operation (usually after each multiplication between polynomial arrays) and to save it in the memory

results during the calculation of the successive polynomial multiplications.
The message/ciphertext buffer has the task of saving the message that will be encrypted or the ciphertext that will be decrypted.
Finally, the decoder, together with the multiplexer, have the task of selecting the right reading of the public polynomial (specifically only those parts of this polynomial are read as will be explained in detail later) to be used during the multiplication, between the two possible memories and to place it in the right place (see the two decoder outputs) to ensure the continuity of polynomial multiplication.
All these characteristics will be described in detail in the subsections of this chapter and in the following one.

4.2.1 Polynomial multiplier

Before outlining the internal architecture of the polynomial multiplier, this subsection illustrates how multiplication between polynomials works.
As explained in Chapter 2, there are many techniques to perform this operation, some complex and other less. This work utilizes one of the easiest among those mentioned, i.e., the Schoolbook method. The algorithm, also called grade school algorithm, is shown below:

Input: Two polynomials $a(x)$ and $b(x)$ in $\mathcal{R}_q$ of degree N.
Output: The product $a(x) \cdot b(x)$ of degree N.
1: $acc(x) \leftarrow 0$.
2: **for** $i = 0; i < N; i = i + 1$ **do**
3: **for** $j = 0; j < N; j = j + 1$ **do**
4: $acc[j] = acc[j] + b[j] \cdot a[i] \bmod \mathbb{Z}_q$
5: **end for**
6: $b = b \cdot x \bmod \mathcal{R}_q$.
7: **end for**
8: **return** acc.

Figure 4.3: Schoolbook polynomial multiplication algorithm

The algorithm receives as input a(x) and b(x), two polynomials belonging to R_q, of degree N = 256. For both polynomials, it is assumed the coefficients of the highest degree of x in the most significant position and those of the lowest degree of x in the least significant position.
The algorithm starts computing the multiplication by multiplying each coefficient of the first polynomial with all the coefficients of the second. The array acc stores the partial values and provides the final result at the end of the polynomial multiplication. Before moving to the next coefficient, an operation called negacyclic shift (instruction 7 of the algorithm) is executed. This involves the rotation to the left of one position of the ring of the second polynomial. In practice, the first coefficient is moved to the

position of the second, the second in the third position and so on. Once the most significant coefficient is reached, it is moved to the least significant position, changed sign. Once the negacyclic shift operation is completed, the second coefficient of the first polynomial will be multiplied with all the coefficients of the second and the negacyclic shift operation will be applied again. The process continues until the last coefficient of the first polynomial is reached. Below, a diagram that summarizes the operation of negacyclic shift is depicted.

$$a_{255}x^{255} + a_{254}x^{254} + ... + ... + ... + ... + ... + a_2x^2 + a_1x + a_0x^0$$

negacyclic shift operation ↓

$$a_{254}x^{255} + a_{253}x^{254} + ... + ... + ... + ... + ... + a_1x^2 + a_0x + (-a_{255})x^0$$

Figure 4.4: Negacyclic shift polynomial operation

In Saber protocol, the polynomial multiplication is always performed between a secret polynomial and a public polynomial. In addition, although secret polynomial is defined on R_q, the range of its coefficients, taking into account all versions of Saber, is contained in a small interval [-5, 5]. Keeping in mind these consideration, [20] proposed their first hardware implementation using 256 multiply-and-accumulate (MAC) in parallel to speed up the execution of the inner loop of the schoolbook algorithm (line 3, 4 and 5) in just one cycle instead of N.

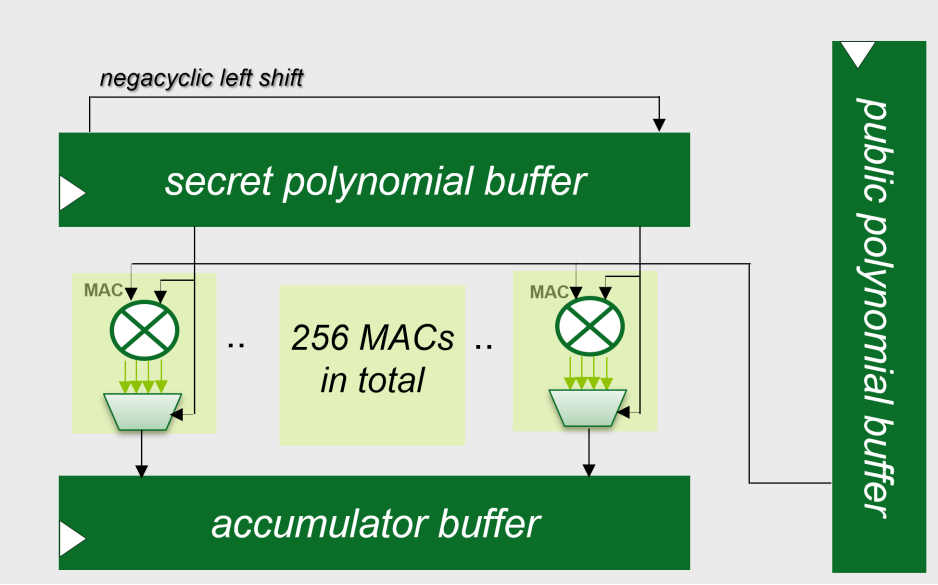

Figure 4.5: Schoolbook polynomial multiplier

Following [20], schoolbook-based polynomial multiplier architecture has four main

components:

- The public polynomial buffer, that loads the first polynomial and provides one of its each coefficient at a time. Public polynomial coefficient are saved using 2's complements representation. Each public coefficient is 13 bits long, so the total size of the buffer is 13x256 bits.

- The secret polynomial buffer, that loads the second polynomial and provides all of its coefficients to be multiplied by the coefficient provided by the buffer described before. Secret polynomial coefficient are saved using Sign and Module (SM) representation. This is an advantage during negacyclic shift because only one bit is flipped to change coefficient sign. Each secret coefficient is 4 bits long, so the total size of the buffer is 4x256 bits.

- The MAC block, which is composed of a multiplier and a multiplexer. It computes one coefficient-wise multiplication and update the accumulator with the new result. The coefficient-wise multiplier is implemented using simple shift and add operations, as shown in Figure 4.6, instead of requiring a true integer multiplier. The multiplier computes up to times-five multiplication to fully support all possible secret coefficient positive interval. Secret coefficient module is used as multiplexer selection signal to select the right result, having performed multiplication using the absolute value of the secret coefficient itself. The accumulator is then updated by adding or subtracting the results depending on the sign-bit of the secret coefficient.

Input: a_i: 13-bit number, s_j: 3-bit number with $0 \leq s_j \leq 5$.
Output: $a_i \cdot s_j$ modulo $q = 2^{13}$.
$r_0 \leftarrow 0$
$r_1 \leftarrow a_i$,
$r_2 \leftarrow a_i \ll 1$,
$r_3 \leftarrow a_i + (a_i \ll 1)$,
$r_4 \leftarrow a_i \ll 2$,
$r_5 \leftarrow a_i + (a_i \ll 2)$,
return r_k, where $k = s_j$.

Figure 4.6: Coefficient-wise shift-and-add multiplier

- The accumulator buffer, that stores the partial values and provides the final result at the end of the polynomial multiplication. Also in this case, the results are stored using 2's complements representation. The buffer size is the 13x256 bits.

Following [20], a further optimization can be implemented to reduce the area occupied by the component.

In the Schoolbook polynomial multiplier implementation, MACs are instantiated in parallel to parallelize the inner loop of the algorithm in Figure 4.3, so all MACs will receive the same public coefficient p_i as one input operand; whereas the other operand (secret coefficient s_j) can be different for the parallel MACs.
Based on this observation, the computation of the public coefficient up to absolute times-five multiplication ($0xp_i$, $1xp_i$, $2xp_i$, $3xp_i$, $4xp_i$, $5xp_i$) of its value, can be centralized for all the parallel MACs. The centralized multiplier forwards the computed multiplies to the parallel MACs. Next, the MAC instances choose their right multiple of a_i depending on their corresponding absolute value of s_j and updates that to the accumulator depending on the sign-bit of the s_j. This approach leads to have just one centralized multiplication, by replacing all the other coefficient-wise multiplier inside a MAC with a simple select operation (e.g. a 6-way multiplexer), thus reducing the area of the MAC unit significantly.
The architecture with the new centralized multiplier optimization is depicted in Figure 4.7.

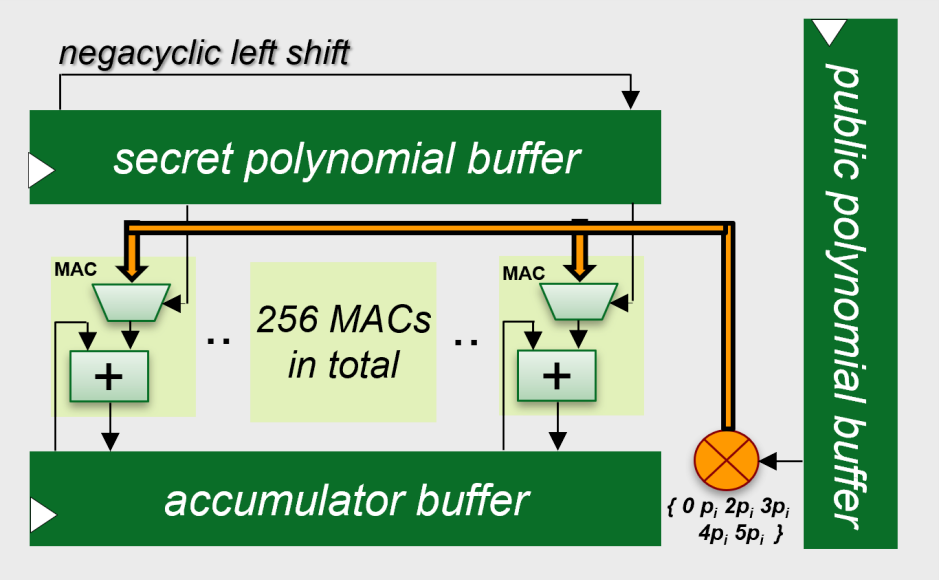

Figure 4.7: Optimization of schoolbook polynomial multiplier

The work of this thesis has as starting point the architecture described, shown in Figure 4.7.
One of the changes made was the use of the Pentium 4 adder [16] inside the MAC block for its optimized structure that allows you to perform the operation faster than basic implementations (i.e., Ripple Carry Adder).
The addition for updating the value of the accumulator register is on 13 bits. To make

sure that it is executed correctly, a Pentium 4 adder based on 12 bits and a full adder, in the most significant part of the final adder, are employed. Using the implementation shown in Figure 4.8 then it is possible to guarantee a correct execution of addition or subtraction based on value on 13 bits in 2's complements representation.

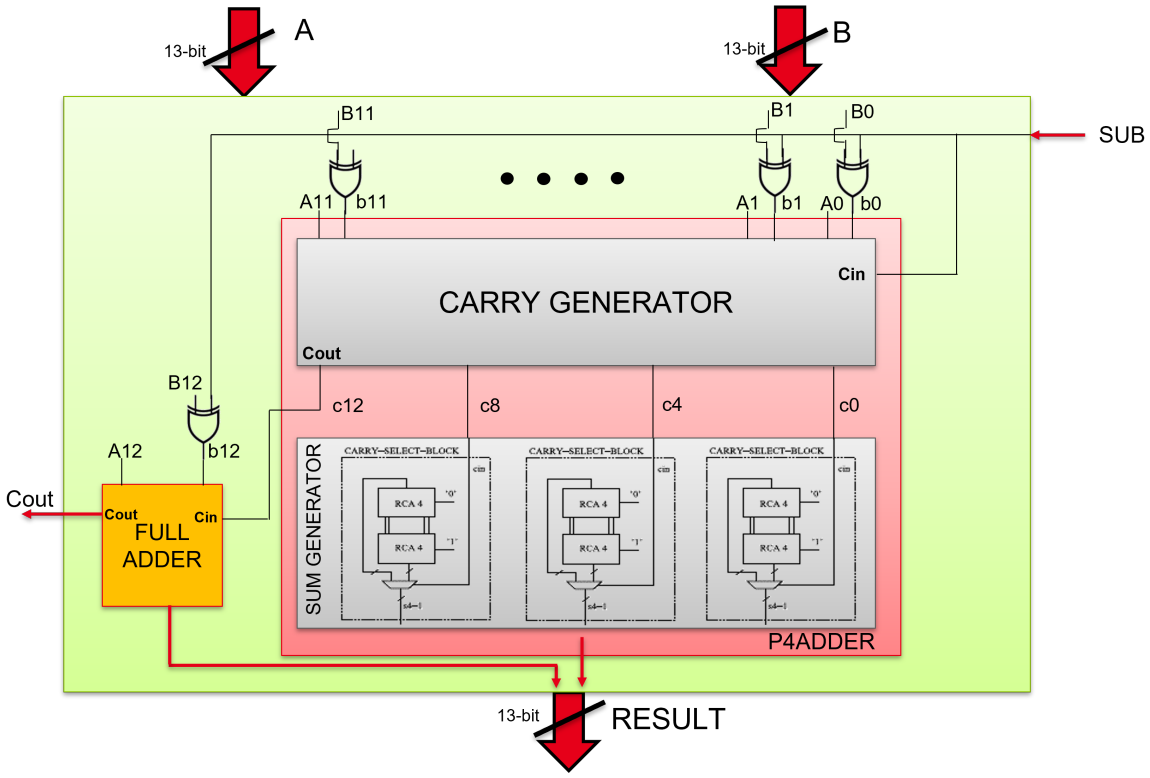

Figure 4.8: Implementation of the adder inside MAC block

The need to perform rounding and multiplication operations based on array or polynomial matrices, requires the modification of the architecture mainly on two blocks, as shown in Figure 4.9.

- **MACs** : Two multiplexers are devised into this element, having two selection signals driven from the FSM controller, to handle the rounding operation. The first multiplexer performs only left shift operations and it is used for adjusting the coefficient m_i of the message polynomial to be encrypted and for the correct representation, depending on the version of saber used, of the coefficient c_i of the ciphertext polynomial to be decrypted. In addition, it is used during the addition of a polynomial constant, which depends on the security level chosen. The second multiplexer instead manages all the final right shift operations for a correct calculation of the final result.

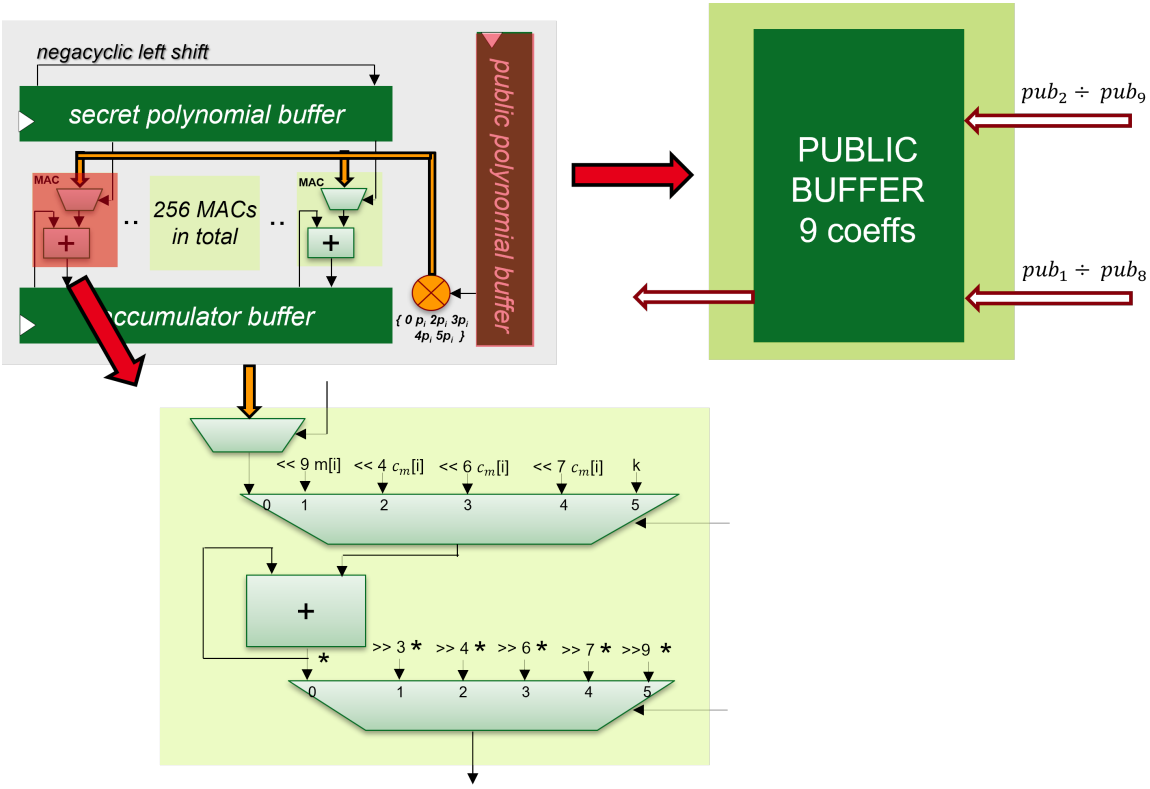

Figure 4.9: Modifications in polynomial multiplier architecture

Constant representation for each coefficient

0	0	0	0	0	0/1	0/1	0/1	0/1	0/1	1	0	0
12	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

Figure 4.10: Constant representation for each polynomial coefficient

Bits positioning	Note
7, 6	**1** if decryption operation in all the versions **0** otherwise
5	**1** if decryption operation in the Saber version 0 otherwise
4, 3	**1** if decryption operation in the Saber or FireSaber versions 0 otherwise

Table 4.5: Bit value of the constant

- **Public buffer** : It has been reduced from 256 to 9 public coefficients polynomial to be stored. This scaling is due to the choice with which the polynomials have

been saved in the memories, in fact it avoids waiting for the entire reading of the public polynomial before it is utilized during the multiplication operation. This choice also involves area saving in the polynomial multiplier. As it is possible to see from the Figure 4.9, the component has two inputs, able to write 8 coefficients in parallel in the buffer. The input at the bottom is used when the coefficients are to be immediately processed in the multiplication operation, whereas the second input (on top) is used to ensure continuity of the execution of the operation when, in the buffer, only the last coefficient left to be consumed.

4.2.2 Memory design

So far it is described how the multiplication between two polynomials is calculated but, one of the aspects that affects its performance, is the storage of data to be processed. In this subsection, it is introduced each of the three memories contained in the datapath, distinguishing their use and the different types of data, including their organization, stored inside. All three memories are designed to work with each of the different versions of Saber. Therefore, for an exhaustive representation, it will be considered the FireSaber version ($l = 4$) for the description of the memories.
As it is possible to see later, there will be a memory that is reused for saving data that will later be consumed in polynomial multiplication. Finally this memory space will be overwritten to contain the designated result in the memory design itself.

Memory are described in the following:

- **Secret key memory** : it has the task of memorizing the array of the secret polynomials **s**. It is chosen as static RAM (SRAM) memory where the reading address is driven by a 6-bit output counter carefully managed by the component controller. Let consider this array structured as

$$s = \begin{bmatrix} s_1 & s_2 & s_3 & s_4 \end{bmatrix}$$

each of its secret polynomials is stored, in little endian notation, as blocks of 16 polynomial coefficients, 4 bit long, on 16 consecutive memory addresses. The total length of each row is 64 bits.

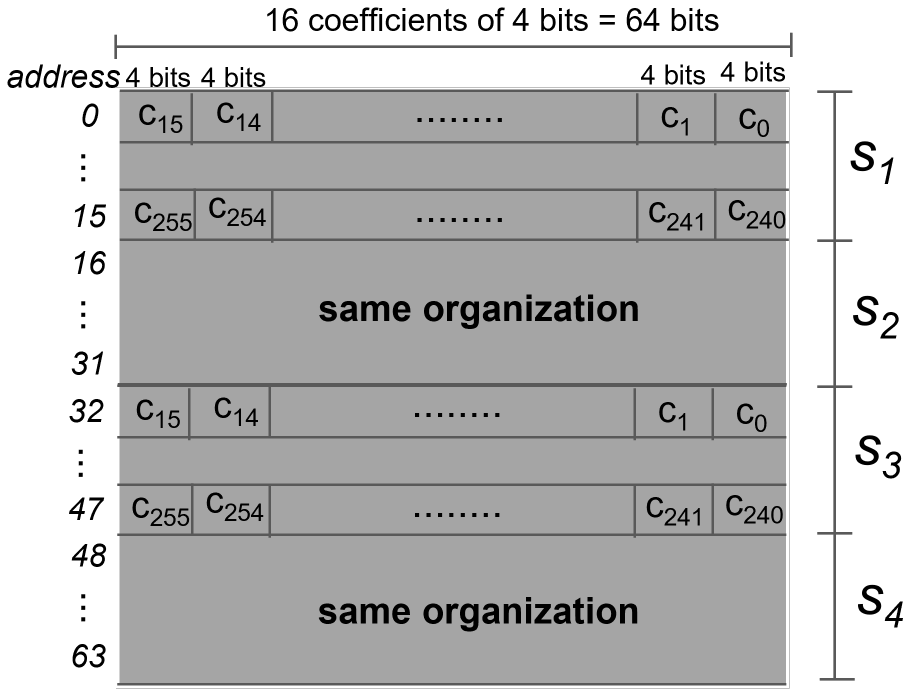

Figure 4.11: Secret key memory

- **Public key memory** : it has the task of memorizing the matrix of the public polynomials **A**. It is chosen as SRAM memory where the reading address is driven by a 9-bit output counter carefully managed by the component controller. Let consider this matrix structured as

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

each of its secret polynomials is stored, in little endian notation, as blocks of 8 polynomial coefficients, 13 bit long, on 32 consecutive memory addresses. The total length of each row is 104 bits.

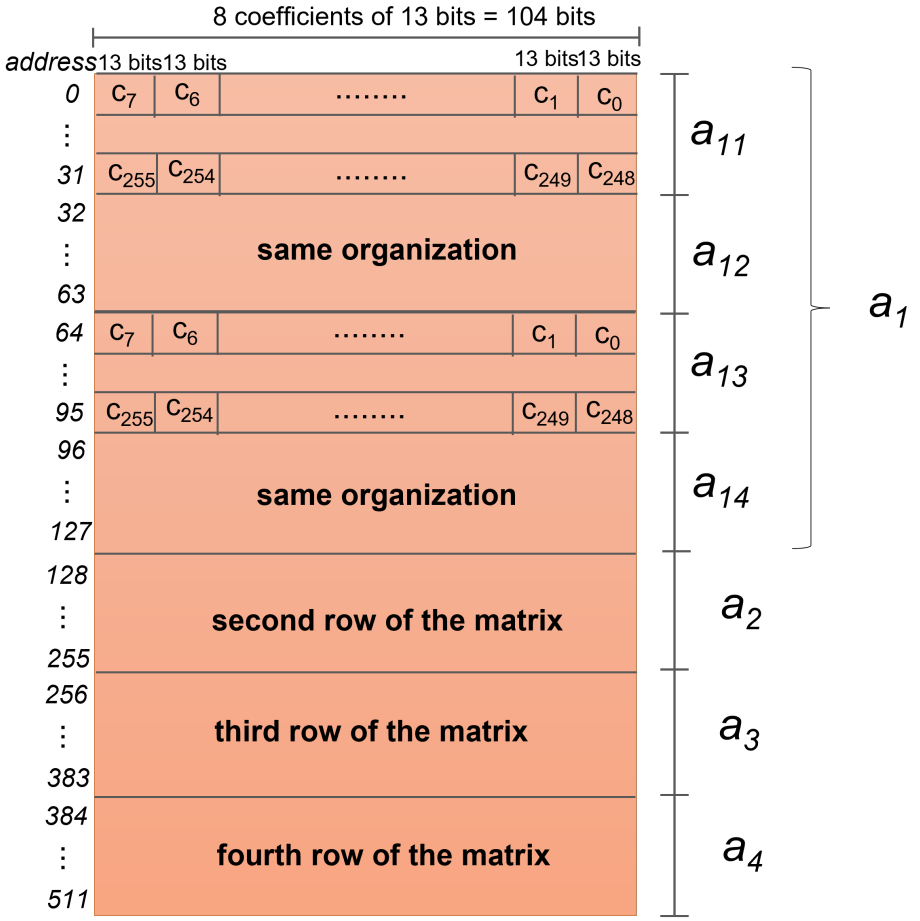

Figure 4.12: Public key memory

- **Result key memory** : it is a SRAM memory and it has the task of memorizing the results of each public key encryption functions, in particular the public polynomial vector **b**, as result of both Key Generation and Encryption functions, the encrypted message of Encryption function and the decrypted ciphertext of Decryption one. The last two results are devised to be stored at the same memory position. Although the dimensions of the coefficients of the latter vary depending on the Saber protocol used, their storage is still on 10 bits.
  In addition, the memory space reserved for saving the result array **b** is used for storing the same array received as input in both encryption and decryption functions. This reuse leads to a saving of space in the sizing of the memory described in the previous point, as it should be the memory containing all public polynomials to be involved during the processing of arithmetic operations.
  For this reason, it is chosen as read/write memory where both the reading and writing address are driven by a 8-bit output counter carefully managed by the component controller.

Let consider this matrix structured as

$$b = \begin{bmatrix} b_1 & b_2 & b_3 & b_4 \end{bmatrix}$$

each of its public polynomials is stored, in little endian notation, as blocks of 8 polynomial coefficients, 10 bit long, on 32 consecutive memory addresses. The total length of each row is 80 bits.

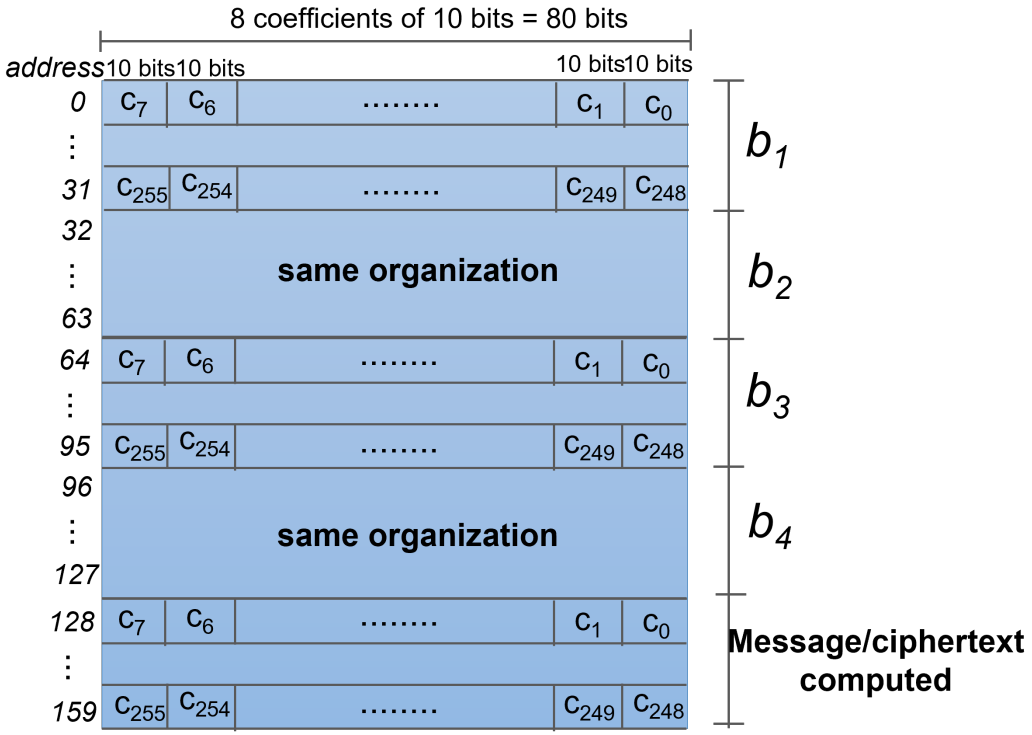

Figure 4.13: Result key memory

4.2.3 Decoder

As described in the previous subsection, the blocks of public polynomial coefficients to be used for computing polynomial multiplication can be read either from public key memory, in the case of the matrix **A**, or from the result key memory, in the case of array **b**. Moreover, the latter has its coefficients on 10 bits, this is a problem because during the operation a length of these coefficients of 13 bits is required. This problem is solved by adding 3 zeros padding in the most significant part of the individual coefficients, adapting them to have the same length of the first input of the multiplexer. It will then select the next octet of public coefficients to be sent to the polynomial multiplier for being processed. This choice depends on the PKE function wished to run. The decoder instead deals with positioning the block of public coefficients in two possible positions inside the public buffer of the polynomial multiplier. The output at the bottom is used

when you want to start the multiplication operation, that is to load the first block of coefficients, while the top output deals with placing the next blocks of coefficients at the highest part of the public buffer at the moment when the last coefficient of the previously loaded block is going to be processed. This latter operation is essential to ensure the correct continuity with which public coefficients are consumed within the multiplier. The choice of decoder and multiplexer output is driven by signals sent by the controller.

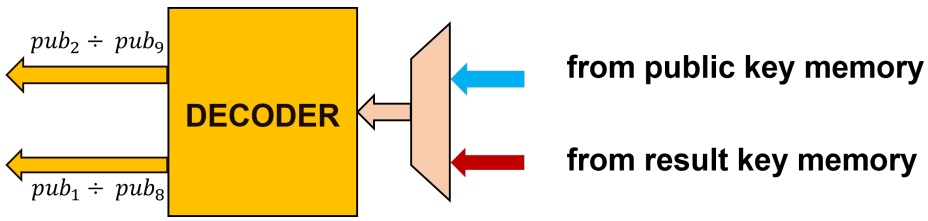

Figure 4.14: Decoder component

4.2.4 Buffers

These memory buffers, composed of registers whose values can be shifted, are fundamental to speed up the operations of reading the polynomials stored by the secret memory and the operations of writing the results, calculated by the polynomial multiplier, in result key memory. There are two buffers in the datapath that have this specific use:

- **Memory buffer** : it is tasked with receiving a block of 16 secret coefficients at a time as input. This operation, if repeated 16 times will lead to the memorization of a complete secret polynomial. The output of this buffer consists of a single port with a length of 1024 bits. They are directly connected to the secret buffer inputs inside the polynomial multiplier to write, before the start of the operation, each secret coefficient of length 4 bits that makes up the secret polynomial itself.

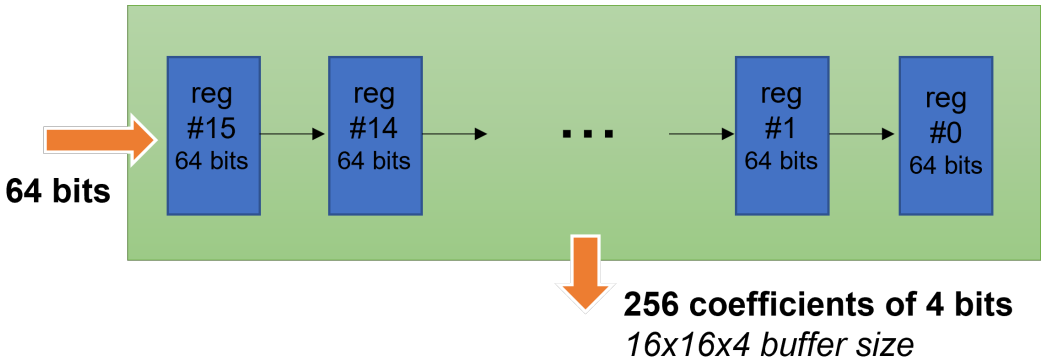

Figure 4.15: Memory buffer structure

- **Result buffer** : it receives as input all the coefficients of the polynomial, calculated within the polynomial multiplier, in blocks composed of eight coefficients each. Moreover, being a sliding buffer, it can be used, during the calculation of the next multiplication, for storing the single block in memory results. The resulting polynomial will be fully saved after 32 writes in memory. This buffer is fundamental because it avoids the introduction of latency times in the storage of the result before it can start to execute the next polynomial multiplication foreseen in the arithmetic operation.

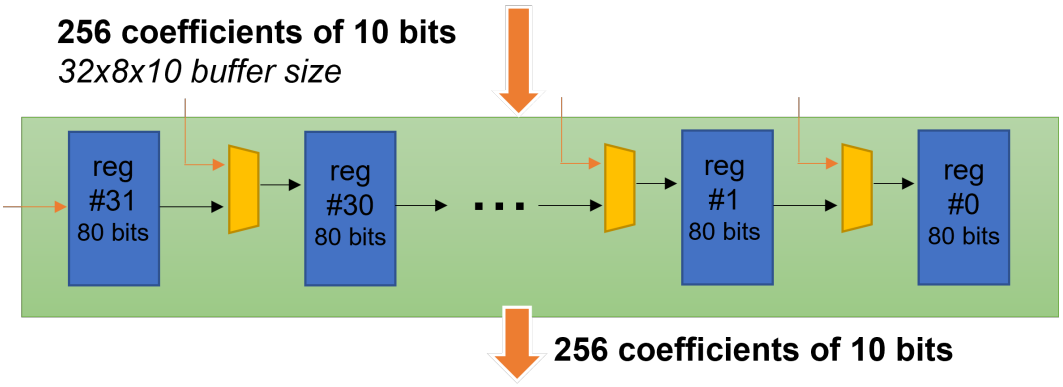

Figure 4.16: Result buffer structure

The last buffer in the datapath, positioned centrally at the top, has the task of containing the message to be encrypted or the cithertext to be decrypted. It consists of an entire 1536 bit long register. It can be seen as a polynomial having 256 coefficients, 6 bit long each. This length is oversized and represents the size of the coefficients of the ciphertext polynomial in the FireSaber version of the protocol.

4.3 Finite State Machine

Most of the complex digital design require an accurate engine to synchronize data consumed inside the Datapath, this task is done by using a controller programmed from a Finite State Machine.
This work provides an implementation of a Moore FSM, shown in Figure 4.17, containing 39 states able to manage all the security level of the Saber protocol. In addition, it is optimized in order to perform as many operations as possible inside each state.

The controller shown in the figure has different colors associated with different operations carried out within the datapath. The states in blue represent the beginning and the end of the Finite State Machine where no operation is performed, while the states in red are responsible of recognizing the type of operation and the version of Saber to be used. They also have the task of loading the first secret polynomial, into the memory buffer, and the first public coefficient octet ready to be loaded into the polynomial multiplier for its processing. The states in green are responsible for reading and loading the coefficients of public polynomials and secret polynomial to be processed. These states are also designed to save the polynomial results in the result key memory during the processing of subsequent polynomial multiplication. The blank state recognizes the type of rounding operation to be performed, specifically the gray states perform this operation in the Key Generation/Encryption operating mode (when matrix-array multiplication is involved), those in yellow for the Encryption and, finally, those orange for the Decryption. The last state in dark green saves the last result of the polynomial multiplication on which the rounding operation was performed.

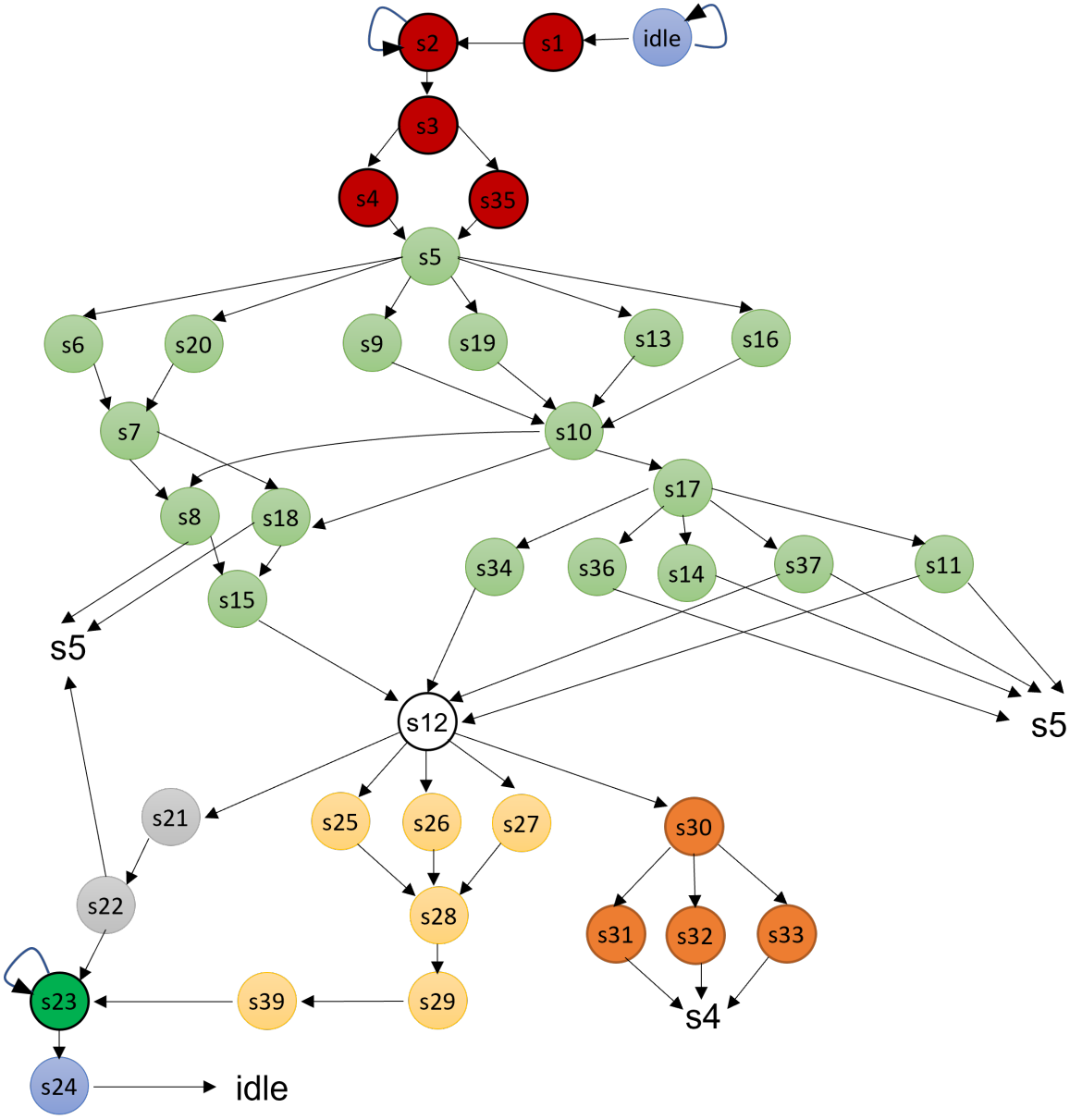

Figure 4.17: FSM controller

In the explanation of the controller operation it is supposed that all the memories are loaded with the polynomial structures useful for the correct execution of the PKE functions that it is wanted to execute. A description of the operations that each state performs on the datapath is provided in the Appendix.

CHAPTER 5

Operating mode

This chapter aims to explain, by using some figures, how the PKE functions are executed by the component described in this work. Also, for space reasons, operations will be only illustrated for the Saber version, $l = 3$, of the protocol.
Moreover, it is assumed that the polynomials involved in each of the functions, described below, are already stored into the correct memories and that thus the component can immediately begin operations processing and subsequent calculation of the results.

5.1 Key Generation

This function has to perform a matrix-array polynomial multiplication of transposed public polynomial matrix A^T, stored inside public key memory, and a secret polynomial array $\mathbf{s}$, stored inside secret key memory, in order to compute the public array $\mathbf{b}$ that will be stored inside result key memory. Before illustrating steps to achieve the desired result, in this section it is briefly revised how matrix-array polynomial multiplication is computed for Saber version of the protocol. Let define A^T and $\mathbf{s}$ as:

$$A^T = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$s = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix}$$

The result $\mathbf{b}$, ignoring rounding operation, is computed as:

$$b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_{11}s_1 + a_{12}s_2 + a_{13}s_3 \\ a_{21}s_1 + a_{22}s_2 + a_{23}s_3 \\ a_{31}s_1 + a_{32}s_2 + a_{33}s_3 \end{bmatrix}$$

Now it is time to illustrate how the key generation function is performed in Saber version.
In the idle state the architectural buffers are initially empty, as shown in Figure 5.1.

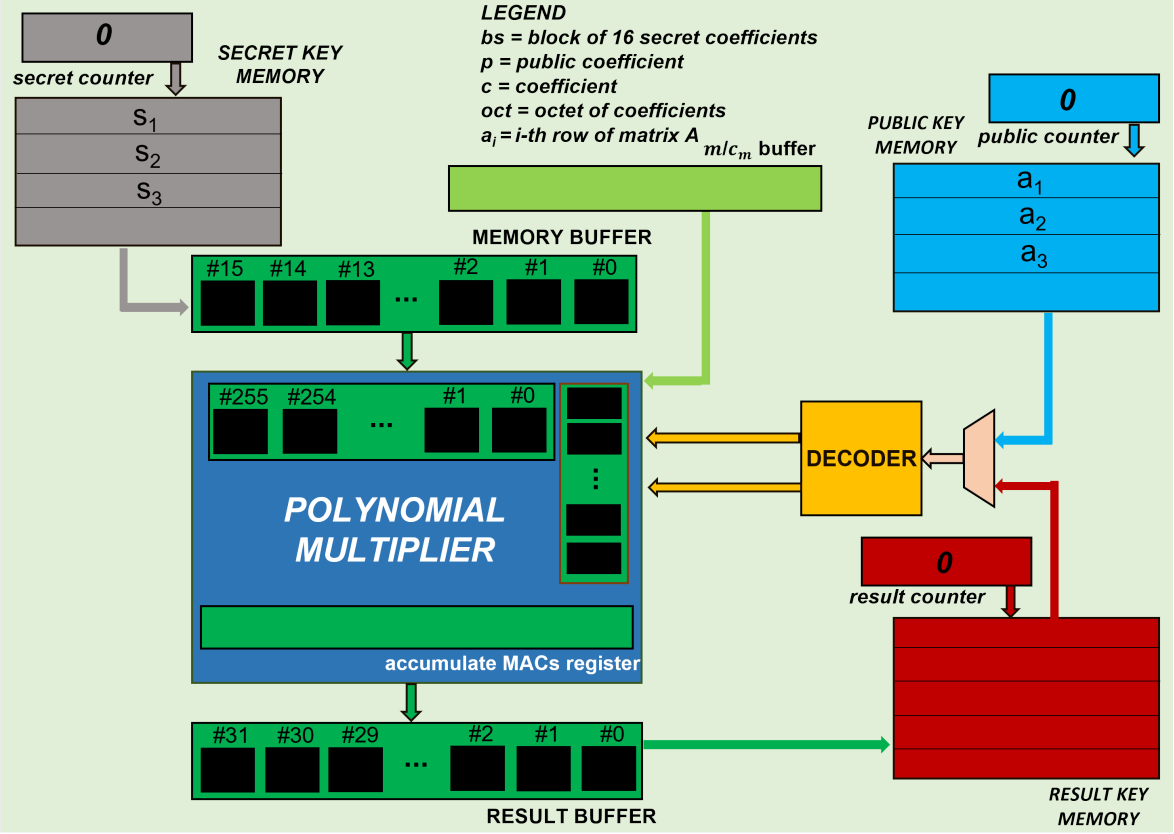

Figure 5.1: Key generation function : architecture buffers empty

The controller recognizes the Key Generation function as a function to execute (S1 state), reads the first secret polynomial s_1 in blocks of 16 coefficients at a time, by updating the secret counter of secret key memory after each read operation, and writes it into the memory buffer (S2 and S3 states). It then reads the first octet oct_1 of coefficients of the public polynomial a_{11} from the public key memory and loads both the newly read octet and the secret polynomial (state s4) into the polynomial multiplier buffers. In this situation, shown in Figure 5.2, the polynomial multiplier is ready to perform the first multiplication.

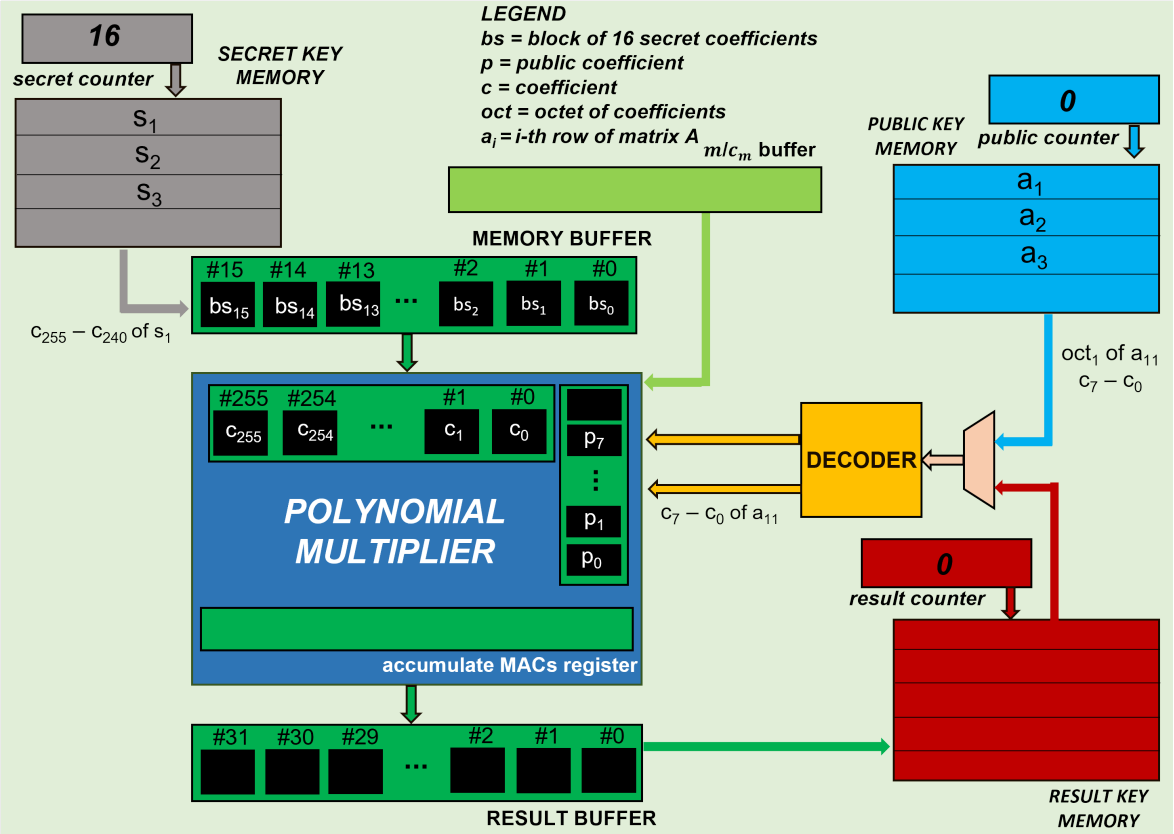

Figure 5.2: Key generation function : starting first polynomial multiplication

At this point the controller begins the execution of the multiplication by multiplying each public coefficient by all the secret coefficients present in the secret buffer. At the end of this operation, the negacyclic shift operation is applied and the public buffer is shifted to allow to the next public coefficient, contained in the buffer, to be executed. The partial result of this operation is accumulated in the internal MAC registers of the polynomial multiplier. This operation is performed four times (state S5) and then the next octet of public coefficients from the public key memory is loaded (S6, S7, S8 states), as shown in Figure 5.3.

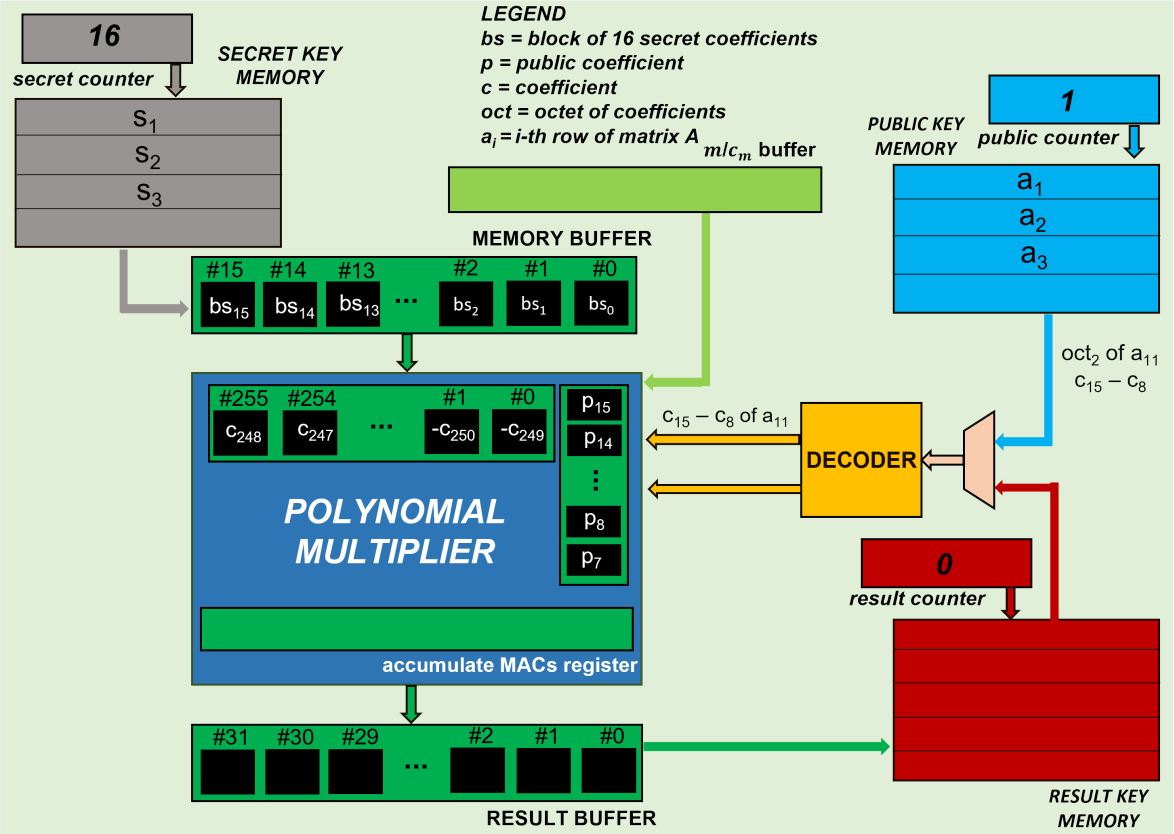

Figure 5.3: Key generation function : loading of second public coefficients octet

The subsequent octets of coefficients are loaded and consumed in the same way as shown in the figure above. The completion of a single polynomial multiplication occurs using 32 of these octets. However, since the component will have to perform the next polynomial multiplication between the second secret polynomial and the second public polynomial, $a_{12}s_2$, during the loading of the last 16 octets of the public coefficients, the controller starts loading the following blocks of 16 secret coefficients in the memory buffer (S9, S10, S8 states). This means that reading overhead costs can be reduced as soon as the multiplier has completed the first multiplication; it may immediately begin to calculate the result of the next operation. Multiplication results between arrays are all accumulated in internal MAC registers. The described situation is illustrated in Figure 5.4.

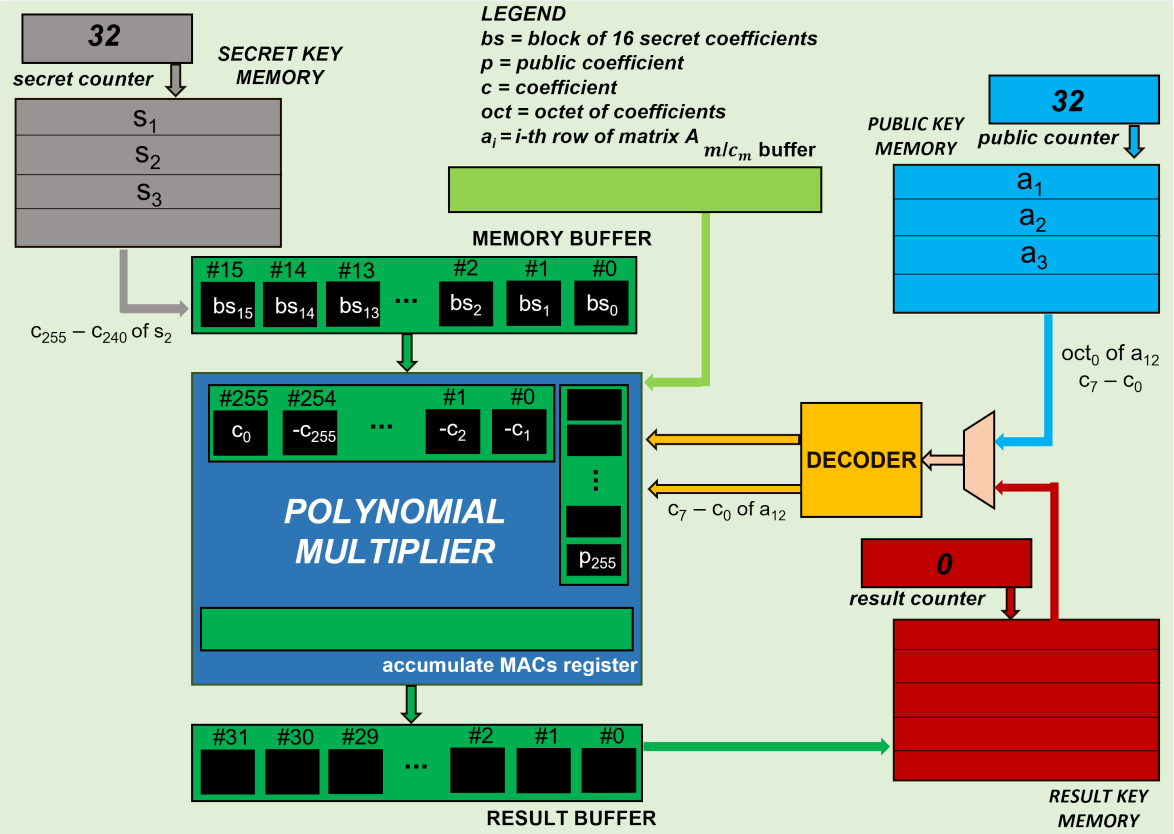

Figure 5.4: Key generation function : loading second secret polynomial during the use of the last 16 octets of the public coefficients

The second polynomial multiplication $a_{12}s_2$ is performed in the same way as the first one. The third secret polynomial s_3 involved in arrays multiplication is loaded into the memory buffer during the use of the last 16 octets of public coefficients in the polynomial multiplier. At the end of the second polynomial multiplication however, the controller resets the secret counter (S10, S17, S14 states) to reload the first secret polynomial s_1 that will be consumed in the first multiplication of the second arrays multiplication. This operation is done during the execution of the third multiplication $a_{13}s_3$. Once this last multiplication is completed, the controller will proceed with the rounding of the result coefficients contained in the MAC registers (S12, S21, S22 states). The rounded result polynomial b_1 will be loaded in the result buffer and the controller will proceed with the execution of the next arrays multiplication to calculate b_2, starting from $a_{21}s_1$ polynomials already loaded inside polynomial multiplier buffers. The situation is shown in Figure 5.5.

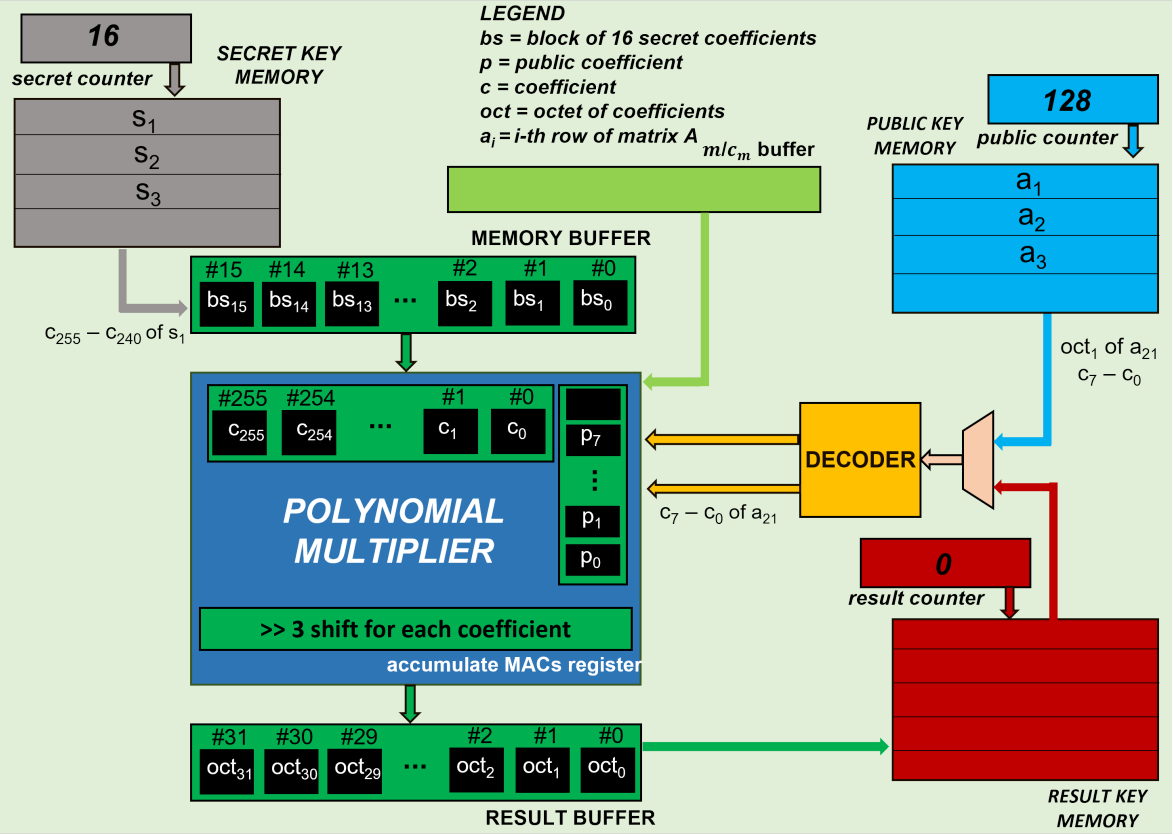

Figure 5.5: Key generation function : completed first arrays polynomial multiplication with rounding operation

At this point the second polynomial multiplication between arrays a_2 and s is performed. This operation is managed in the same way as the first multiplication $a_1 s$ explained so far. Compared to the previous case, the result buffer is full, in fact it contains the previous result of the vector polynomial multiplication b_1 already rounded. The controller, in this case, besides reading the blocks of the secret coefficients during the execution of the polynomial multiplication, must also save the 32 blocks contained in the memory buffer in the result key memory. This operation is accomplished during the last polynomial multiplication $a_{23} s_3$ of the arrays multiplication $a_2 s$. At every public coefficients octet reading from the public key memory, the controller performs a writing of a block in the buffer result, starting from the address indicated in the counter result of the previous Figure 5.5 (S16 state). Moreover, in the last 16 octet readings, the controller, in addition to writing the blocks in the result buffer, reads the first secret polynomial (S13 state) ready to be used in the last arrays multiplication $a_3 s$ that will be executed later. At the end of the last polynomial multiplication $a_{23} s_3$, the rounding operation is handled and the new final result b_2 is loaded into the result buffer. The provided description is referred to Figure 5.6.

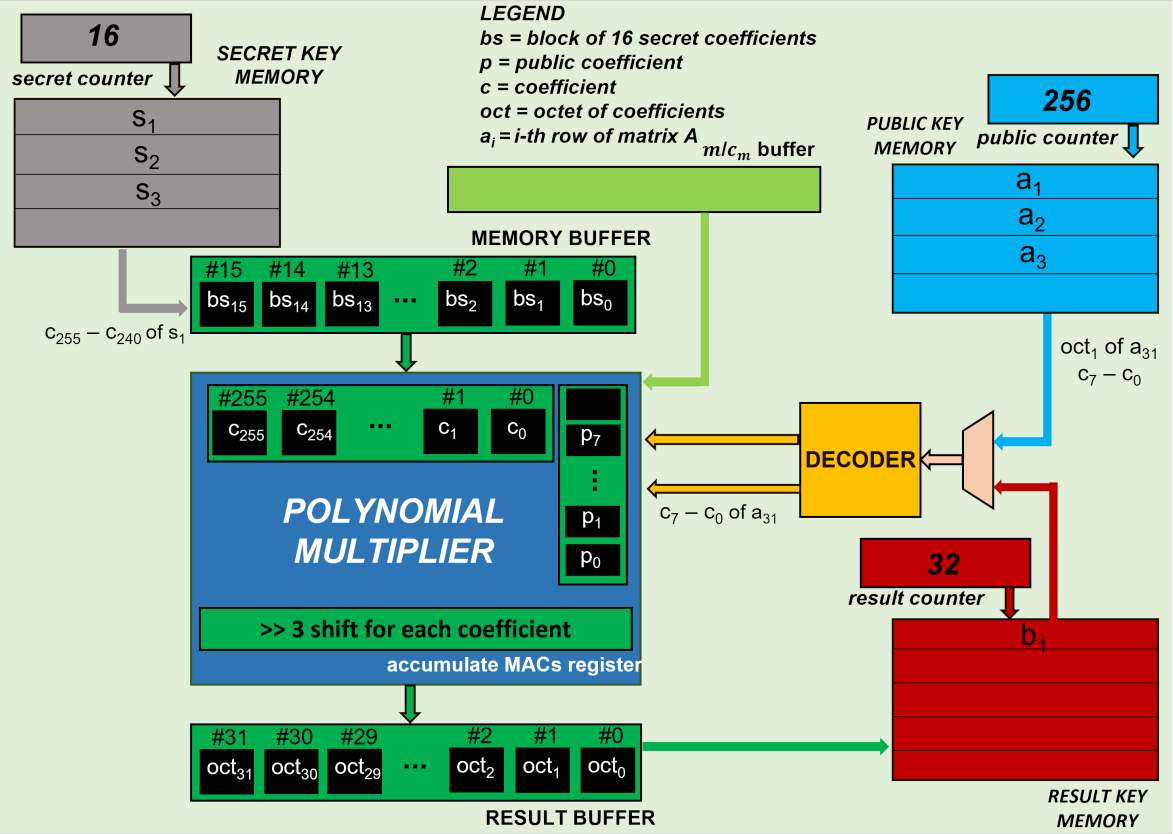

Figure 5.6: Key generation function : completed second arrays polynomial multiplication with rounding operation, previous result are stored inside result key memory

To complete the correct execution of the key generation function, the polynomial arrays multiplication $a_3 s$ is performed in the same way as described in the previous page. After the end of the last polynomial multiplication $a_{33} s_3$ and its rounding operation, the final polynomial result b_3 is saved in the memory buffer. The controller will manage its write to the result key memory through the S23 state, executing 32 consecutive writes of the blocks contained in the memory buffer to the corresponding address in the result counter. At the end, as shown in Figure 5.7, the polynomial array **b** is completely stored in memory. The controller cleans all the registers and buffers of the component and returns to the situation shown in Figure 5.1.

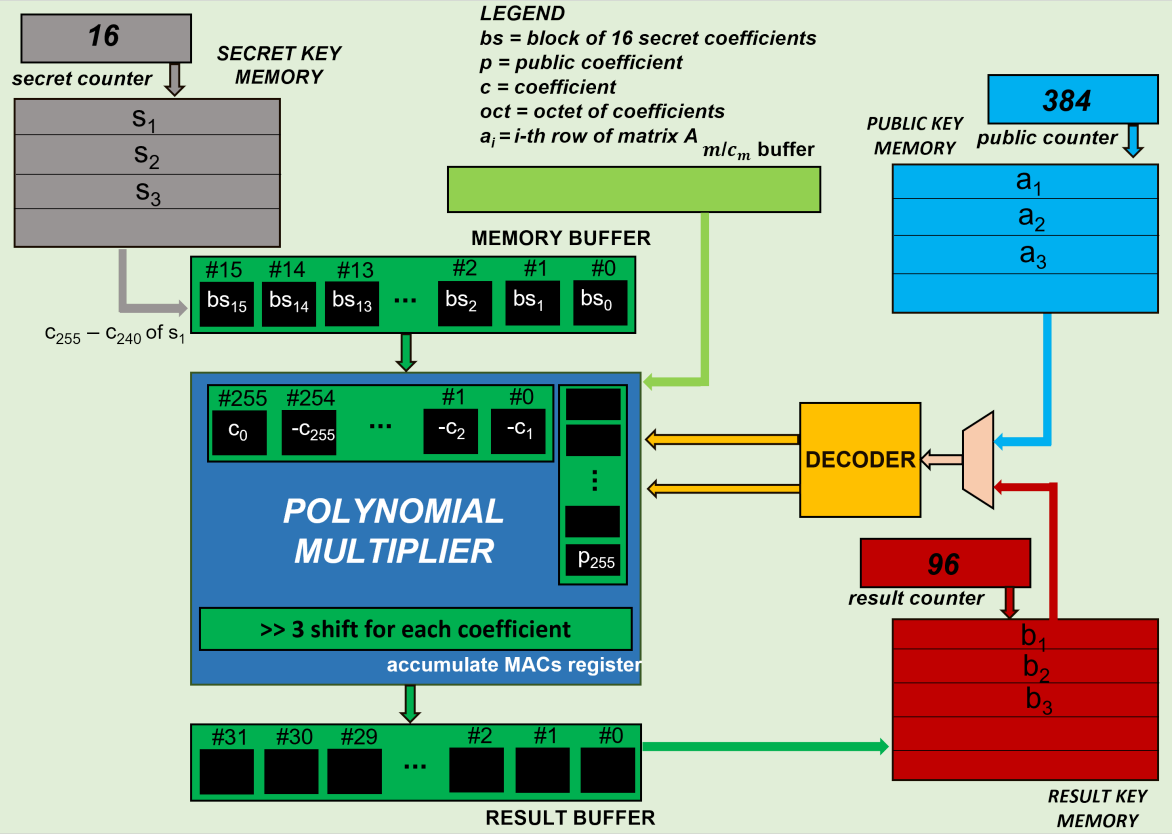

Figure 5.7: Key generation function : completed last arrays polynomial multiplication with rounding operation, polynomial results are stored inside result key memory

5.2 Encryption

This function has to perform firstly a matrix-array polynomial multiplication between a public polynomial matrix **A**, stored inside public key memory, and a secret polynomial array **s**, stored inside secret key memory, in order to compute the public array **b'** that will be stored inside result key memory. Finally the function has to compute the polynomial multiplication between a public transposed polynomial array b^T, stored inside result key memory, and the same secret polynomial array **s** used for the first operation. Before illustrating steps to achieve the desired result, in this section it is briefly revised how array-array polynomial multiplication is computed for Saber version of the protocol. Let define b^T and **s** as:

$$b^T = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}$$

$$s = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix}$$

The result **v**, ignoring rounding operation, is computed as:

$$v = b_1 s_1 + b_2 s_2 + b_3 s_3$$

In order to facilitate the design of the controller, the following PKE function has been chosen to calculate as the first operation the array polynomial multiplication between the public polynomial b^T and **s**, then polynomial multiplication where the matrix of public polynomials **A** is involved.

Now it is time to illustrate how the encryption function is performed in Saber version.
In the idle state the architectural buffers are initially empty. The starting of the operation is also denoted by the introduction of the message to be encrypted in the message/ciphertext buffer, as shown in Figure 5.8.

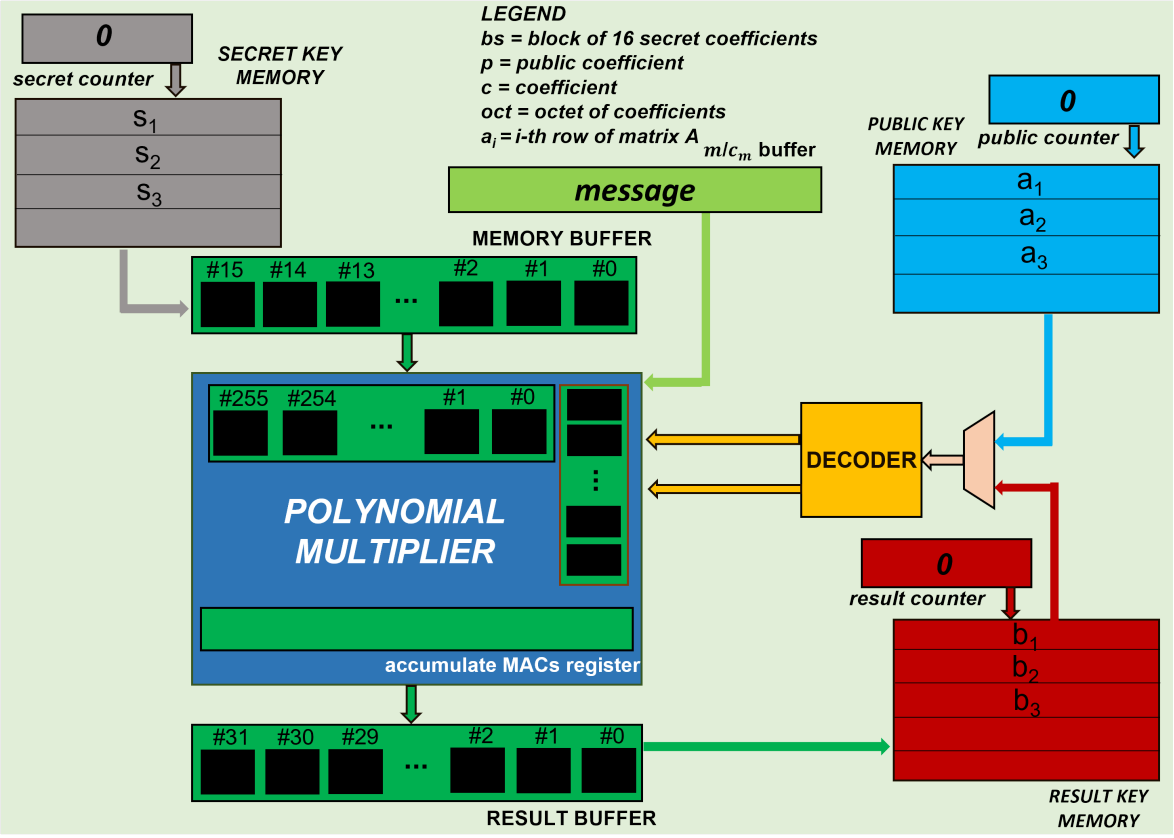

Figure 5.8: Encryption function : Memory and buffer contents before the execution starting

Polynomial multiplication arrays occurs in the same way as in the key generation function for the first row a_1 of the matrix A^T and the secret polynomial array s. In this case, however, the polynomials of the public array are saved in the result key memory

and not in the public one, so the public coefficients octet loading must be handled differently by the controller. When performing a single polynomial multiplication contained in the resulting polynomial v, the controller loads the new octets to public coefficients using the S20, S7 and S18 states, while loading the next secret polynomial, during the execution of the operation itself, it occurs through the use of the S19, S8 and S18 states. Having clarified these slight changes, Figure 5.9 shows the loading of the coefficients of the polynomials involved in the first polynomial multiplication $b_1 s_1$.

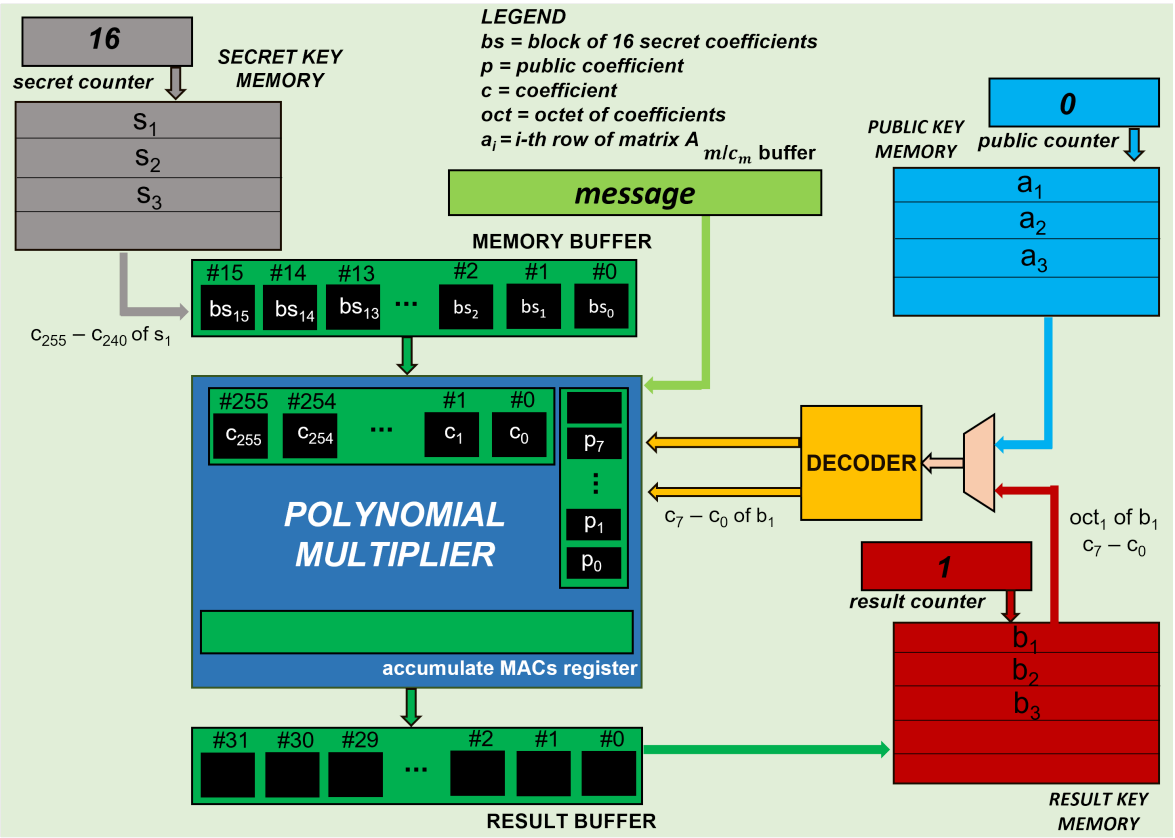

Figure 5.9: Encryption function : beginning of the first polynomial multiplication

The polynomial multiplications $b_2 s_2$ and $b_3 s_3$ are performed according to the guidelines described in the previous section and the variations in the reading flow of the octets of public coefficients mentioned above. At the end of the three polynomial multiplications, the result is contained in the registers inside the MACs in the polynomial multiplier. The rounding phase, in the case of the Saber version, is done by adding the coefficients of the result itself with the polynomial constant h_1, S30 state, and subtracting the coefficients of the message, contained in the message buffer/ciphertext, shifted nine positions to the left, S32 state. The polynomial obtained after this operation is saved in the memory buffer, the resulting counter will be updated with the value to which the controller will have to write the resulting polynomial contained in the memory buffer. Figure 5.10 shows the situation just described for easy understanding.

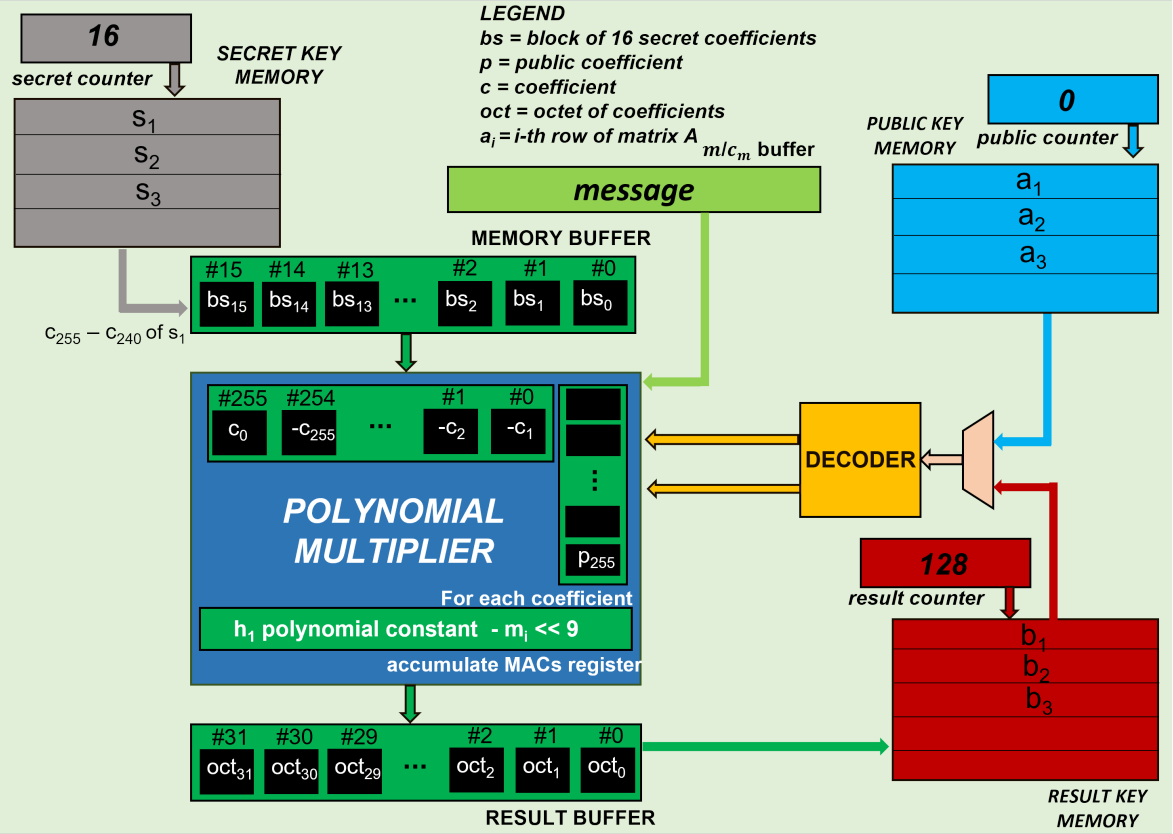

Figure 5.10: Encryption function : end of the array polynomial multiplication with rounding operation and result polynomial inside resutl buffer

At this point, the controller proceeds with the execution of the multiplication between the matrix of public polynomials **A** and the secret array of polynomials **s**. Also here, the procedure for the execution of this operation to be followed is identical to that seen in the previous section with two main changes. The first concerns the writing of the 32 blocks of the coefficients of the result, that in Figure 5.11 will be called ciphertext, in the result key memory beginning from the address indicated in the counter result of Figure 5.10. The writing of these blocks will be done during the entire execution of the first polynomial multiplication $a_{11}s_1$ by using S13 and S16 state. The second concerns the zero resetting of the result counter at the end of the polynomial multiplication $a_{11}s_1$ itself, in the S34 state, to ensure that the polynomial results calculated from now on are saved from the correct memory address in the result key memory.

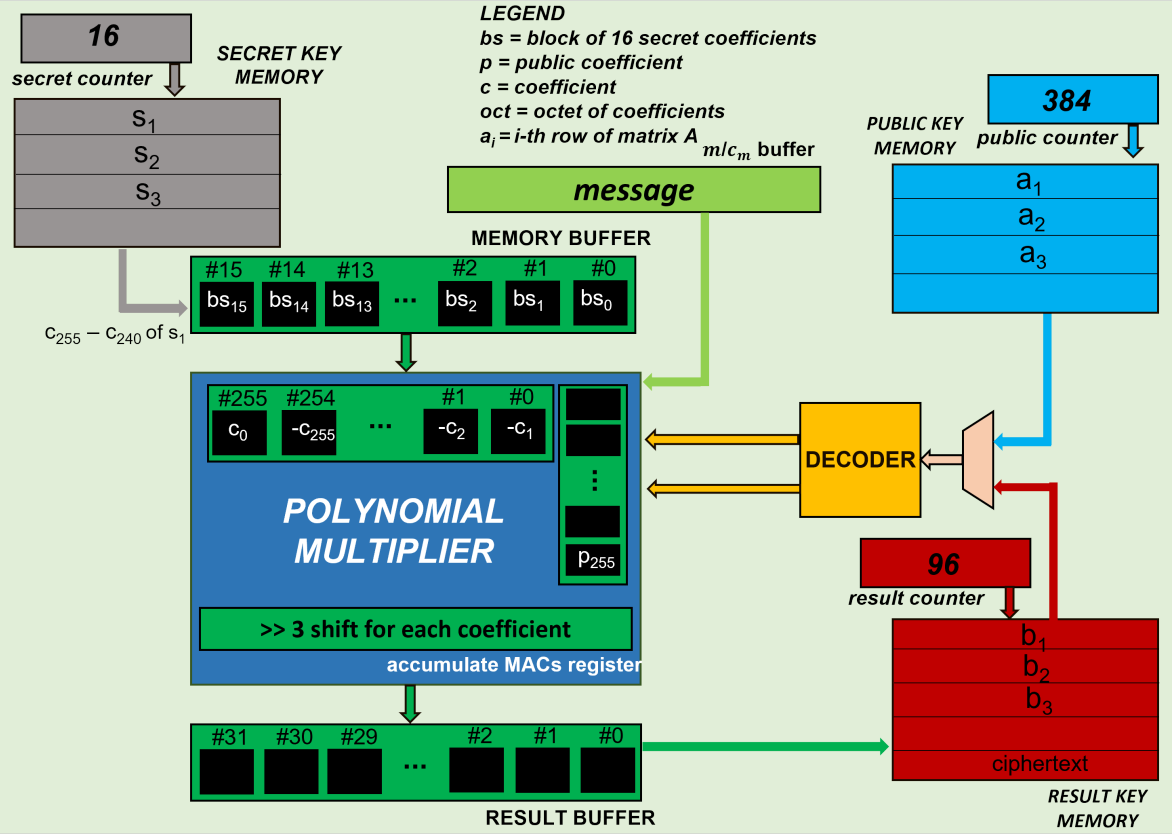

Figure 5.11: Encryption function : end of executing function with all the results written inside result key memory

5.3 Decryption

This function has to perform a array-array polynomial multiplication between a public transposed polynomial array **b**, stored inside result key memory, and a secret polynomial array **s**, stored inside secret key memory, in order to compute the public array **v** that will be stored inside result key memory after the rounding operation.

Now it is time to illustrate how the decryption function is performed in Saber version.

In the idle state the architectural buffers are initially empty. The starting of the operation is also denoted by the introduction of the ciphertext to be decrypted in the message/ciphertext buffer, as shown in Figure 5.12.

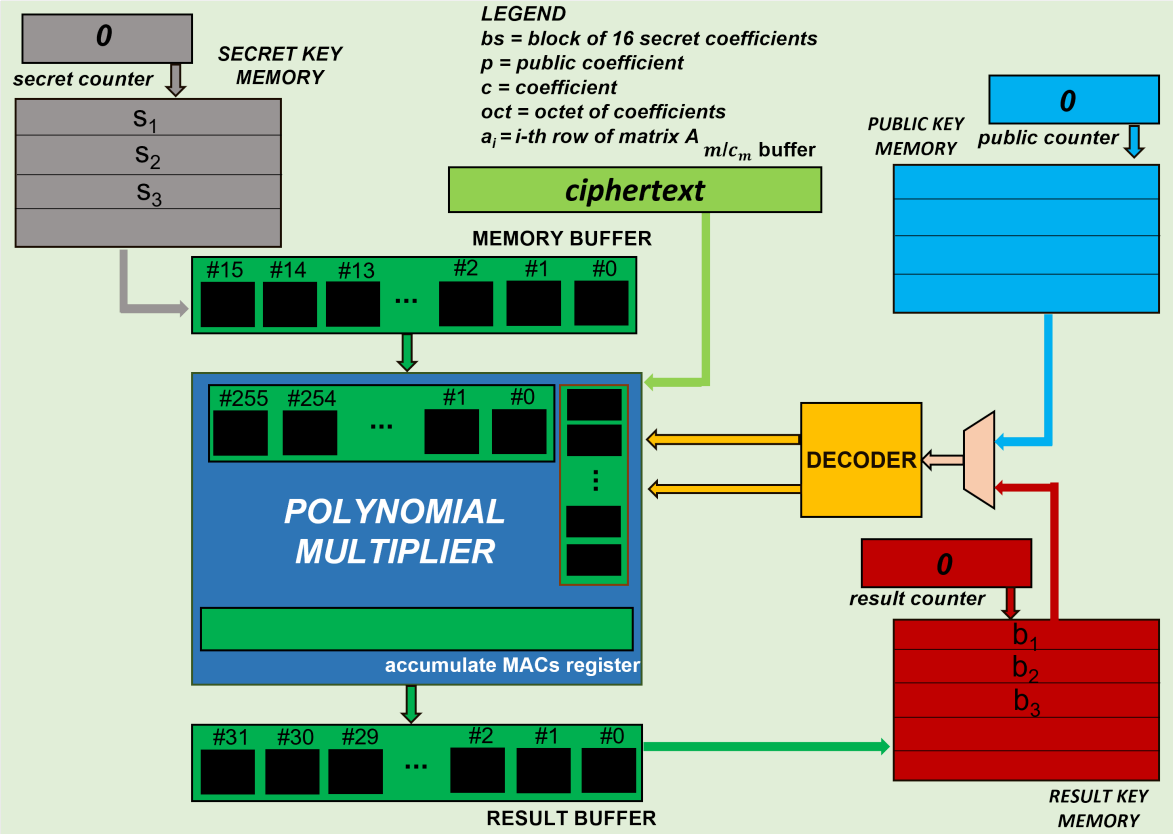

Figure 5.12: Decryption function : Memory and buffer contents before the execution starting

The execution of the polynomial multiplication occurs exactly as that seen during the encryption function, both the polynomials to be loaded and the octet flow of the public coefficients correspond exactly. The only variation is in how the rounding operation, in the Saber version of the protocol, is performed. Once the arrays polynomial multiplication is finished, the result coefficients are subtracted with ciphertext coefficients, in input, shifted by six positions to the left, S26 state. The resulting polynomial is also summed with the polynomial constant h_2 in the S28 state. The final result is loaded in the result buffer, S29 state, and then saved in the result key memory starting from the address 128. These last operations are controlled by the controller in S29 state.

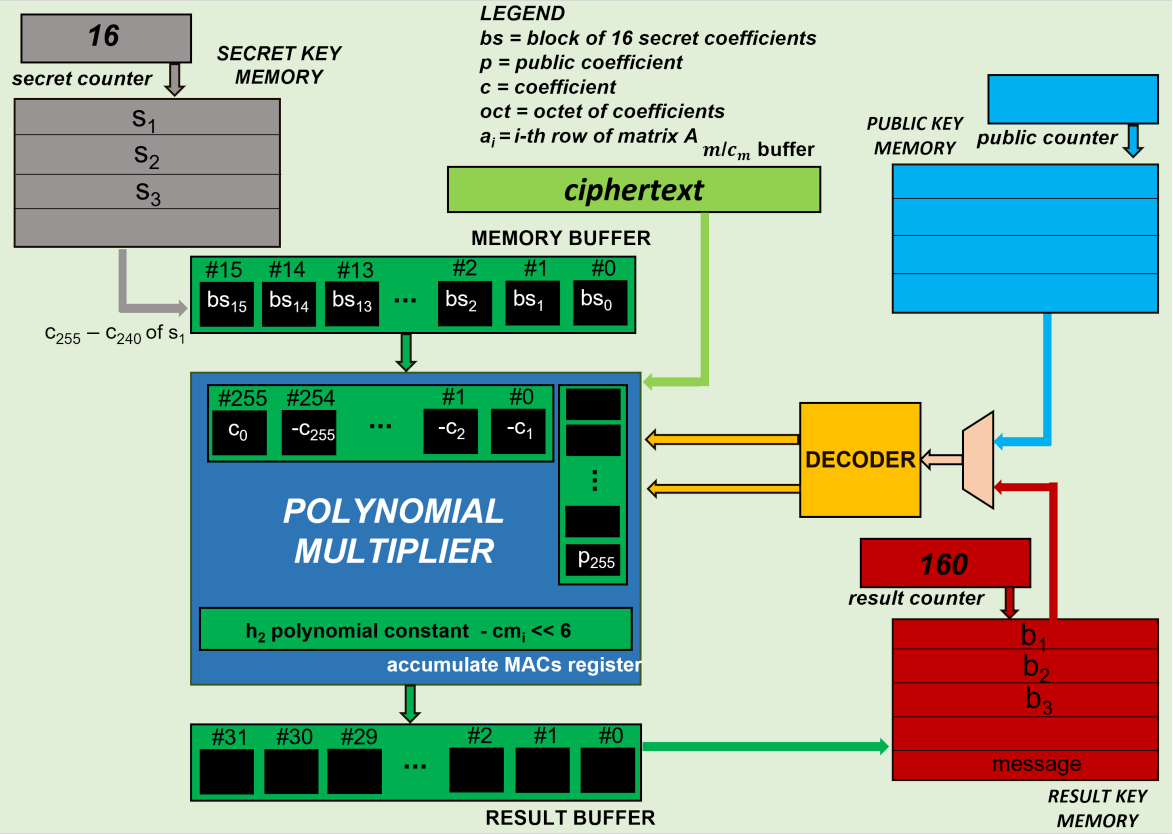

Figure 5.13: Decryption function : end of executing function with message result written inside result key memory

CHAPTER 6

Verification

Design verification is the most important aspect of the product development process and its intent is to verify that the design meets the system requirements and specifications. The approach for this phase involves generating test vectors to be used as input values of the component described in this work. The produced results will be checked through a golden model stimulated using the same test vectors.
In this chapter is firstly introduced the golden model used during the verification phase of this work. Then it is given a brief description of its implementation and finally are described the changes made so that it can be adapted to the correct implementation of this phase.
In the second section, it is described the implemented toolchain aiming at verifying the consistency of the results produced by the golden model and the component described in this work. It is also explained how test vectors are generated and the method used to estimate the design verification coverage.

6.1 Golden model

The golden model [25] chosen for the verification part is a SW implementation of the KEM Saber protocol functions. It relies on Toom Cook's algorithm for performing polynomial multiplication. It also presents libraries for both the generation of polynomial structures and the packaging of data that are widely exploited during the execution of KEM functions.
However, some changes are made to ensure proper use by the toolchain :

- **KEM functions organization** : KEM functions have been transformed into PKE ones, based on the right generation of polynomials involved in each of them.
- **Polynomial structures generation** : the generation of both matrix and arrays polynomials takes place through initial seeds, 32 bytes long, saved in rows, one for each PKE function.

  - *seed_KeyGen.txt* : it contains the seeds for the generation of the corresponding matrix of public polynomials **A** and array of secret polynomials **s**.
  - *seed_Encr.txt* : it contains the seeds for the generation of the corresponding matrix of public polynomials **A**, array of secret polynomials **s**, array of public polynomials **b** and polynomial message m to be encrypted.
  - *seed_decr.txt* : it contains the seeds for the generation of the corresponding matrix of public polynomials **A**, array of secret polynomials **s** and polynomial ciphertext cm to be decrypted.

- **Polynomial structures writing** : a library is designed to write all polynomial structures generated and the computed results, inside the PKE function, in files. In particular, they are written according to the memory format explained in section 2.2 of chapter 4. As a result of this operation, two different files are produced for each polynomial structure: the first contains the polynomial coefficients in the classical decimal form, the second instead writes their values in binary, using either notation in 2'complement or in module and sign. For more details on representative coefficient choices, see subsection 4.2.1.

6.2 Toolchain

The toolchain is a simply shell script, written in Bash command language, that coordinates all the operations useful for a correct execution of the verification phase. the script, to run correctly, must receive three inputs:

- **Operating mode**

Operating mode input	Description
KG	Key Generation
E	Encryption
D	Decryption

Table 6.1: Toolchain script : operating mode input

- **Saber version**

Saber version input	Description
LS	LightSaber
S	Saber
FS	FireSaber

Table 6.2: Toolchain script : Saber version input

- **N** value : it represents the number of tests to be performed during the verification phase.

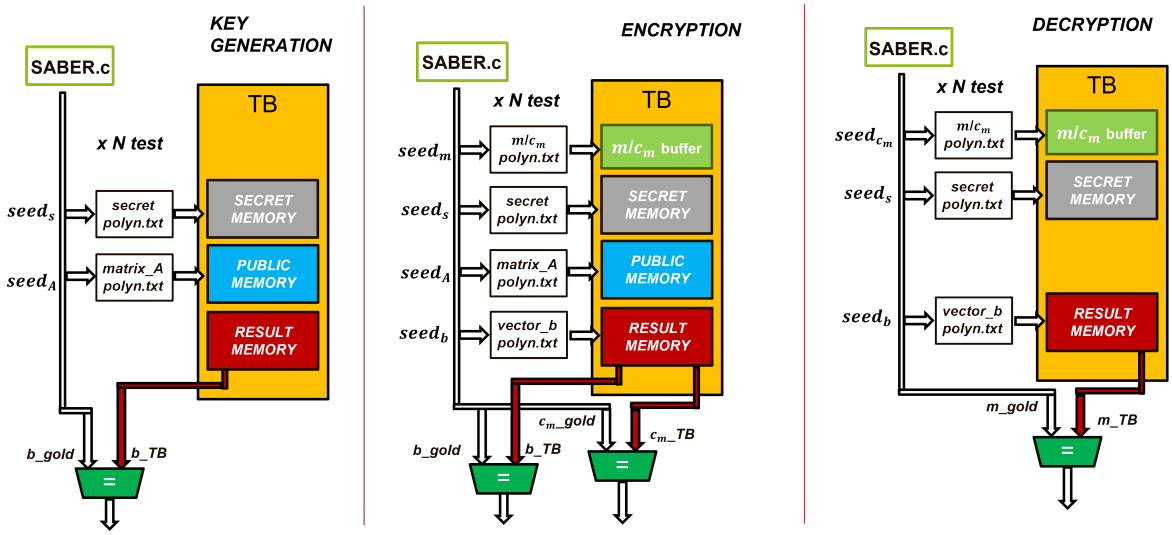

Figure 6.1: Verification scheme for each PKE function

Before running the toolchain, the seed file corresponding to the operation to be performed must be written. Once the command line input parameters are defined, the toolchain perform some operation described below.
As a first step, the user writes the initial seeds value to be used in the operating mode it is wanted to verify. The script initially creates folders where the files described in "Polynomial structures writing" item must be created. The component source files are compiled and the same thing is done for the golden model. First of all the execution of the golden model is launched, which will generate both the input files, to be loaded in the testbench, and the output files useful for the final comparison of the results obtained. Input test vectors are generated by using a software function implementation of Shake128, which is an algorithm to generate unique hash value starting from an input value. This function receives as input the seed file corresponding to the selected operating mode and, for each of the seed value contained in the file, it generates the polynomial structures involved in the operation. Then the simulation of the component is also launched, by choosing the test bench written specifically to run both the Saber version and the operating mode set in input. Depending on the operating mode to be performed, files containing polynomial structures are loaded into the memories of the component. At the end of the simulation, the test bench will have generated one or more output files containing the results just calculated. At this point the results are compared; the successful outcome of this operation will allow the script to continue with the next test. Before doing that, the increments of the seeds used for the selected operating mode, is performed using a C program written with this specific reason. This increase allows to generate different polynomial structures at each test, making

the verification phase more efficient. Once all the tests are finished, a log file will appear. It contains all the tests carried out with the relative results of comparison. A brief illustration of the operations of the toolchain is schematized in Figure 6.1.

6.2.1 Verification result

As described before, input seeds are 32 byte long and each operating mode requires more than one seed to generate the proper polynomial structures for well verifying the component functionality. This approach leads to the difficulty of performing an exhaustive component verification for each protocol version and operating mode. In fact, the number of combinations of test vectors is so high that it would take weeks or even months to run all of them. Unfortunately, since the development time of this work was limited, it was decided not to follow this approach for design verification.

Given the difficulties described above, it was decided to run a number of tests equal to N=200,000 for each of the available operating modes, using each security level of the Saber protocol. Seed values, inside files, are initialized all to zero and incremented by one at the end of each individual run of the selected operating mode. The total execution time was 48 hours and verification results reports a success of all test vectors generated, although these are not sufficient to ensure the full functioning coverage of this work.

```
#!/bin/bash

N=200000

./toolchain_verif.sh KG LS $N
./toolchain_verif.sh KG S $N
./toolchain_verif.sh KG FS $N

./toolchain_verif.sh E LS $N
./toolchain_verif.sh E S $N
./toolchain_verif.sh E FS $N

./toolchain_verif.sh D LS $N
./toolchain_verif.sh D S $N
./toolchain_verif.sh D FS $N
```

CHAPTER 7

Results

The hardware design architecture of this work is written using the VHDL description language. The compilation and simulation of the component is performed by the Electronic Design Automation (EDA) tool QuestaSim multi-language environment. Finally, the proposed architecture is synthesized using a 22nm library from Global Foundaries (GF) based on the planar process of Fully-Depleted-Silicon-On-Insulator technology (FD-SOI). The library is called GF22FDSOI and the synthesis phase is repeated using different clock periods in order to find maximum working frequency.
This chapter is split in two sections where different results are described. The first one aims at showing the performance and power consuming of the polynomial multiplier building block by performing a single polynomial multiplication, while the last one concerns the complete working architecture. At the end of each section, results are compared with existing state-of-the-art implementations by differentiating hardware, software and HW/SW codesign solutions.
The evaluation of the results is based on two different metrics:

- **Timing performance** : it represents the number of clock cycles needed to perform either a single polynomial multiplication or a given PKE function. This parameter is retrieved from QuestaSim during simulation phase.
- **Power dissipation** : it represents the power dissipated during the execution of a given Saber operating mode or a single polynomial multiplication. These values are taken from synthesis reports.

7.1 Single polynomial multiplication

The first analysis of performance and power consumption is executed on the single polynomial multiplication, by isolating the polynomial multiplier building block contained in the component architecture. The synthesis phase is performed with different clock periods. The results lead a maximum operating frequency of 768 MHz and a power consumption of 24,1 μW/MHz.
Some state-of-the-art implementations are used as comparison with this work in Table 7.1 and, for each of them, results are shown referring as the same working frequency for a better comparison analysis.

	Technology	Frequency [MHz]	# of cycles	Execution time [μs]	Power dissipation
Reference [24]	HLS 65nm	500	-	-	66.4mW
This work	GF22FDSOI	500	256	0.51	10.93mW
Reference [21]	40nm CLN40G	400	-	-	1.15mW
This work	GF22FDSOI	400	256	0.64	8.82mW
Reference [22]	40nm CMOS	300	380	1.26	176.7mW
This work	GF22FDSOI	333	256	0.77	7.40mW

Table 7.1: Result comparison for one single polynomial multiplication

This work leads an 83% of power consumption saving compared with [24], which uses schoolbook algorithm to implement polynomial multiplication. It computes its result using a synthesizer in high-effort mode, 65nm technology based. The selected effort makes the synthesis process longer but it allows to find better results
Instead, an increase of 86% of power consumption is shown respect the HW implementation of [21] which saves 90% of multiplication by applying an efficient optimization on Karatsuba algorithm. The comparison result used for [21] is estimated taking into account the reported 39mW average of power consumption during the execution of PKE functions. For the Saber version (l=3) of the protocol, only 35.5% of this power is deployed for the total four array polynomial multiplication operations in the Encryption function. So, a value of 1.15 mW of power consumption is estimated in the cited work for a single polynomial multiplication.
The third comparison is done with an HW implementation of Rounding-Learning-With-Error, based on the NTT. Using this approach, input polynomials need an NTT transform before the execution of a coefficient-wise multiplication, and an INTT transform to bring the result in the starting polynomial domain. As reported in [22], the NTT transform is performed in 160 clock cycles (cc) with a power consumption of 58.9mW. Multiplication takes only 20 cc and the final INTT transform is done in 200 cc. Considering all these values and an equal power consumption for both transformations, the computed results of power dissipation and execution time are reported in

Table 7.1. This work leads to advantages from both the two metrics used for the results evaluation. A single polynomial multiplication is performed using a 33% reduction of clock cycles. Considering the use of all transformation, the power consumption saving is 95% with the execution time reduced of 39%.

7.2 PKE functions

Although the previous section described the performance and power consumption of polynomial multiplication alone, the most relevant results derive from the synthesis of the complete architecture of this work. The produced reports illustrates a maximum working frequency of 430 MHz with a power consumption of 4.28mW. Given the difficulty of finding power consumption data in state-of-the-art implementations, the widely used comparison parameter in this section is the performance time, expressed in clock cycles, for the completion of each of the PKE functions for each version of the Saber protocol. Following, the section firstly introduced comparison with a software implementation of PKE functions and finally, the same thing will be done for some of the most relevant hybrid HW/SW solutions.

	Protocol version	Key Generation cycles	Encryption cycles	Decryption cycles
Reference [23]	Saber	695,547	875,874	180,327
This work	LightSaber	1083	1600	567
	Saber	2371	3145	824
	FireSaber	4173	5204	1081

Table 7.2: Results : comparison with software implementations of PKE functions

In Table 7.2, it is clear how a HW polynomial multiplier accelerator can significantly speed up the execution of PKE functions, in fact this work is 100x faster than the solution proposed in [23].

In Table 7.3 are instead shown result comparisons having the same order of magnitude. Although this work is initially inspired by [20], their results comparison lead to a maximum improvement of timing performance of 12% for Saber version of the protocol. The Saber version results of [20], do not take into account rounding operations and polynomial multiplication storing. Furthermore, by looking at the execution time results, there is an even more remarkable improvement of every PKE functions, up to 45% for Encryption of Saber version. This is due to the different maximum working frequency of the works involved in the comparison.
As in the case of single polynomial multiplier analysis, [21] counts a lower numbers of clock cycles necessary to complete functions, up to 65% of clock cycles overhead during the Encryption execution of the FireSaber security level. On the other hand, in the last section it is reported that the 35.5% of average 39mW power consumption is employed for array polynomial multiplications. In this work, even if power results are not so reliable, the average working power consumption is only 4.28mW.

	Technology	Frequency [MHz]	Protocol version	KeyGen cc/μs	Encr cc/μs	Decr cc/μs
Ref [20]	UltraScale+ FPGA	250	Saber	2645/ 6.58	3592/ 13.37	892/ 3.37
Ref [21]	UltraScale+ FPGA	100	LightSaber	519/5.19	664/6.64	326/3.26
			Saber	943/9.43	1156/11.56	408/4.08
			FireSaber	1531/15.31	1811/18.11	490/4.90
This work	GF22FDSOI	430	LightSaber	1083/2.52	1600/3.72	567/1.32
			Saber	2371/5.51	3145/7.31	824/1.92
			FireSaber	4173/9.70	5204/12.10	1081/2.51

Table 7.3: Results : hardware and HW/SW comparisons with PKE functions

CHAPTER 8

Conclusion

In this work, a high-performance and flexible component architecture for lattice-based public-key cryptography is proposed, with Saber PKE as a case study. The component was designed to manage all the arithmetic operations involved in the PKE functions, in particular polynomial multiplications and rounding operations. The flexibility of the component derives from the possibility to perform all PKE operations for LightSaber, Saber and FireSaber. The core of this architecture is the polynomial multiplier. Following [20], it uses the schoolbook multiplication algorithm with some optimizations employed for achieving high speeds, while remaining simple and highly scalable. A single polynomial multiplication is performed in only 256 cycles. Some memory buffers are introduced to speed up the loading of polynomials to be consumed in the polynomial multiplier, to synchronize the readings of the new polynomials from the memories and the writing of the multiplier results in the memory. Synthesis results show a working frequency of 430 MHz and some advantages in timing performance and power consumption with respect to other state-of-the-art implementations. On the other hand, there is an implementation solution [21] in which the use of Karatsuba polynomial multiplication algorithm, with some optimizations, lead to more performing results. This information is however useful to understand that the characteristics with which the Saber protocol has been designed (i.e., modular structure and powers of two moduli) allow to find more and more solutions that simplify the architecture and improve performance.

Bibliography

[1] Benioff, Paul (1980). "The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines". Journal of Statistical Physics. 22 (5): 563–591

[2] Feynman, Richard (June 1982). "Simulating Physics with Computers" (PDF). International Journal of Theoretical Physics. 21 (6/7): 467–488.

[3] Preskill, John (2018). "Quantum Computing in the NISQ era and beyond". Quantum. 2: 79

[4] Gibney, Elizabeth (2 October 2019). "Quantum gold rush: the private funding pouring into quantum start-ups". Nature. 574 (7776): 22–24

[5] Rodrigo, Chris Mills (12 February 2020). "Trump budget proposal boosts funding for artificial intelligence, quantum computing"

[6] Frank Arute et al., Quantum Supremacy using a Programmable Superconducting Processor. Nature, 574:505–510, 2019.

[7] National Institute of Standards and Technology. 2015. Post-quantum cryptography

[8] Bernstein, Daniel J. (2009). "Introduction to post-quantum cryptography". Post-Quantum Cryptography. Nature. Vol. 549. pp. 1–14.

[9] Mermin, David (28 March 2006). "Breaking RSA Encryption with a Quantum Computer: Shor's Factoring Algorithm". Physics 481-681 Lecture Notes. Cornell University

[10] Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM Journal on Computing, 26(5):1484–1509, Oct 1997.

[11] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER. Proposal to NIST PQC Standardization, Round2, 2019.

[12] https://classic.mceliece.org/index.html

[13] https://ntru.org/

[14] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehle. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In 2018 IEEE European Symposium on Security and Privacy (EuroS P), pages 353–367, 2018.

[15] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom Functions and Lattices. In EUROCRYPT 2012, pages 719–737, 2012.

[16] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, P. Roussel, Desktop Platforms Group, Intel Corp, "The Microarchitecture of the Pentium 4 Processor", Intel Technology Journal Q1, 2001

[17] Karatsuba, A.A., Ofman, Y.P.: Multiplication of many-digital numbers by automatic computers. In: Doklady Akademii Nauk. vol. 145, pp. 293–294. Russian Academy of Sciences (1962)

[18] Donald Knuth. The Art of Computer Programming, Volume 2. Third Edition. Addison-Wesley, 1997.

[19] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in Toom-Cook multiplication: an Application to Module-lattice based Cryptography. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020(2):222–244, Mar. 2020.

[20] S. Sinha Roy and A. Basso, "High-speed Instruction-set Coprocessor for Lattice-based Key Encapsulation Mechanism: Saber in Hardware," IACR Transactions on Cryptographic Hardware and Embedded Systems, vol. 2020, no. 4, pp. 443–466, Aug. 2020.

[21] Y. Zhu et al., "LWRpro: An Energy-Efficient Configurable Crypto-Processor for Module-LWR," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 68, no. 3, pp. 1146-1159, March 2021, doi: 10.1109/TCSI.2020.3048395.

[22] S. Song, W. Tang, T. Chen and Z. Zhang, "LEIA: A 2.05mm2 140mW lattice encryption instruction accelerator in 40nm CMOS," 2018 IEEE Custom Integrated Circuits Conference (CICC), 2018, pp. 1-4, doi: 10.1109/CICC.2018.8357070.

[23] Bin Wang and Xiaozhuo Gu and Yingshan Yang, "Saber on ESP32", Cryptology ePrint Archive, Paper 2019/1453, 2019

[24] Imran, Malik & Abideen, Zain & Pagliarini, Samuel. (2020). An Experimental Study of Building Blocks of Lattice-Based NIST Post-Quantum Cryptographic Algorithms. Electronics. 9. 1953. 10.3390/electronics9111953.

[25] https://github.com/KULeuven-COSIC/SABER

[26] Alagic, G. , Cooper, D. , Dang, Q. , Dang, T. , Kelsey, J. , Lichtinger, J. , Liu, Y. , Miller, C. , Moody, D. , Peralta, R. , Perlner, R. , Robinson, A. , Smith-Tone, D. and Apon, D. (2022), Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process, NIST Interagency/Internal Report (NISTIR), National Institute of Standards and Technology, Gaithersburg, MD, [online], https://doi.org/10.6028/NIST.IR.8413, https://tsapps.nist.gov/publication/get$_p df.cfm?pub_i d$ = 934458($Accessed July 9, 2022$)

CHAPTER 9

Appendix

9.1 FSM states

- **IDLE (idle state)** : in this state the controller disables datapath buffers, except the one containing the message/ciphertext, cleans the contents of the registers, including address counters, inside the polynomial multiplier and waits until input *start* is raised for to launch a PKE function execution.
- **RECOGNIZE_OP_MODE (S1 state)** : in this state the controller recognizes the PKE function to execute and enables the secret key memory.
- **LOAD_FIRST_SECR_POLYN (S2 state)** : in this state the controller sends the signals to perform a secret key memory reading to the address indicated in the corresponding address counter. The secret counter will then be incremented. The polynomial coefficients read are written in the memory buffer. These operations are performed 16 times in order to have the first secret polynomial read entirely.
- **FIRST_PUB_OCTET (S3 state)** : in this state, the controller checks whether the first octet of public coefficients is to be read from the public key memory, containing the matrix **A**, or from the result key memory, containing the array **b**, during the encryption operation.
- **LOAD_OCTET_PUBMEM_PUB_BUFF (S4 state)** : in this state the controller reads the first octet of public polynomials from the public key memory and loads it into the lowest part of the public buffer inside the polynomial multiplier while, in the secret buffer is loaded the secret polynomial, read in S2, contained in the memory buffer.
- **MANAGE_MULT_OP (S5 state)** : in this state the multiplication between the public coefficient and the secret polynomial is performed, the result is saved in the MACs registers of the polynomial multiplier. Then the secret buffer will

be rotated to apply the negacyclic shift operation and the public buffer will be shifted to allow the next public coefficient to be processed. These operations are repeated four times. In the next states other operations may be performed in parallel with the loading of the new octet of public coefficients to be consumed.

- **NEXT_OCTET_PUB_MEM_NO_OP (S6 state)** : in this state the controller recognizes that will read the new octet of public coefficients from the public key memory, before doing this the public counter is increased so that it is read in the right location in the memory. Meanwhile, the polynomial multiplier continues its execution, by consuming the fourth last public coefficient in public buffer, following the operations described in S5. This state is used during the execution of the array-array polynomial multiplication involving the matrix **A**.

- **CHOOSE_OCTET_POLYN_MULT (S7 state)** : in this state the controller continues performing polynomial multiplication by consuming the third last public coefficient in public buffer. The new octet of public coefficients is read.

- **OCTET_PUBMEM_PUBBUF (S8 state)** : in this state the controller continues the execution of the polynomial multiplication consuming the penultimate public coefficient left in the buffer. It also takes care of loading the new octet, read from the public key memory, in the upper part of the public buffer in order to guarantee continuity to the multiplication in the successive states. The controller also checks if 32 octets have already been read (256 coefficients in total), if this has not happened it returns to S5 to start consuming the coefficients of the new octet. It also checks whether the last octet of public coefficients of a array-array polynomial multiplication has been consumed, If this is the case, a new state will be created to use the last coefficient in the public buffer and then proceed with the rounding operation.

- **NEXT_OCTET_PUB_MEM_SECR_LOAD_OP (S9 state)** : in this state the controller recognizes that will read the new octet of public coefficients from the public key memory, before doing this the public counter is increased so that it is read in the right location in the memory. The controller also checks the number of octets that have already been read. In case the FSM has read at least 16 of them and there are successive polynomial multiplications to be performed (for example, it is not reached the end of an array-array or matrix-array polynomial multiplication), the secret counter is incremented and the secret buffer is enabled. This will allow, in the next state, the reading of a block of 16 secret coefficients to be stored in the secret buffer. In subsequent uploads of the octets of public coefficients (16 times in total) will be read the blocks of the secret coefficients until it will be stored the next secret polynomial fully loaded in the secret buffer. Meanwhile, the polynomial multiplier continues its execution, by consuming the

fourth last public coefficient in public buffer, following the operations described in S5. This state is used during the execution of the array-array polynomial multiplication involving the matrix **A**.

- **END_SINGLE_POLYN_MULT (S10 state)** : in this state the controller checks the end of a single polynomial multiplication between your polynomials by checking if 32 blocks of public coefficients have been read. If not, check the memory from where to load the next octet of public coefficients returning to the states s8 (public key memory) and S18 (result key memory). Meanwhile, the polynomial multiplier continues its execution, by consuming the third last public coefficient in public buffer, following the operations described in S5.

- **END_SINGLE_POLYN_MULT_PUBMEM_8 (S11 state)** : in this state the controller loads the secret polynomial and the new octet of public coefficients. The first, contained in the memory buffer, is loaded inside the secret buffer while the last, which has been read from the public key memory, is loaded at the bottom of the public buffer of the polynomial multiplier. If the current single polynomial multiplication concerned the last multiplication between polynomial arrays, then the controller will continue with the rounding operation in subsequent states, otherwise it will resume the execution of the multiplication operation. Meanwhile, the polynomial multiplier continues its execution, by consuming the last public coefficient in public buffer, following the operations described in S5.

- **CHOOSE_ROUNDING_OP (S12 state)** : in this state, the controller disables the update of the registers inside MACs, so that they keep the result calculated by the polynomial multiplier. Also decides the rounding operation to be performed on this result, taking into account the PKE function and the polynomial structures that it is performing.

- **NEXT_OCTET_PUB_MEM_LOADSECR_STORERES_OP (S13 state)** : this state is used during the matrix-array polynomial multiplication, specifically in the case when the first multiplication between the first row of the matrix and the vector has already been performed. In addition, the controller puts itself in the condition in which, in successive multiplication between arrays, the last polynomial multiplication is being performed. In this case the controller recognizes that it lasts at least 16 public coefficient blocks to complete polynomial multiplication so, it increases the public counter (new public coefficients octet will be read from public key memory), secret counter and the result counter. It also enables the result buffer and the write signal in the result key memory, since in subsequent states it will save the result of the previous polynomial multiplication calculated in the specified memory. The controller will be also in charge of loading new secret coefficients blocks. Meanwhile, the polynomial multiplier

continues its execution, by consuming the fourth last public coefficient in public buffer, following the operations described in S5.

- **END_SINGLE_PMULT_PUBMEM_8_RSTSECRCNT (S14 state)** : in this state the controller loads the secret polynomial and the new octet of public coefficients. The first, contained in the memory buffer, is loaded inside the secret buffer while the last, which has been read from the public key memory, is loaded at the bottom of the public buffer of the polynomial multiplier. The controller recognizes that the current single polynomial multiplication concerned the second last multiplication between polynomial arrays, then the controller resets secret counter of secret key memory. This is due to the fact that, during the last polynomial multiplication involved in the arrays polynomial multiplication, the first secret polynomial will be loaded ready to be used during the next multiplication between arrays. At the end, the controller will resume the execution by proceeding with the last the multiplication operation. Meanwhile, the polynomial multiplier continues its execution, by consuming the last public coefficient in public buffer, following the operations described in S5.

- **LAST_MULT_ARRAY_MULT (S15 state)** : in this state the controller consumes the last public coefficient contained in the public buffer, thus ending the polynomial multiplication.

- **NEXT_OCTET_PUB_MEM_SECR_STORERES_OP (S16 state)** : this state is used during the matrix-array polynomial multiplication, specifically in the case when the first multiplication between the first row of the matrix and the vector has already been performed. In addition, the controller puts itself in the condition in which, in successive multiplication between arrays, the last polynomial multiplication is being performed. In this case the controller, increases the public counter (new public coefficients octet will be read from public key memory) and the result counter, enables the result buffer and the write signal in the result key memory, since in subsequent states it will save the result of the previous polynomial multiplication calculated in the specified memory. The operation will be completed by storing the 32 blocks contained in the memory buffer. Meanwhile, the polynomial multiplier continues its execution, by consuming the fourth last public coefficient in public buffer, following the operations described in S5.

- **BEFORE_LOAD_OCTET_LOW_PART (S17 state)** : in this state the controller, at the end of a single polynomial multiplication, can decide several states. It can choose to reset the result counter if the multiplication between the first row of the matrix and the array in PKE encryption has been completed. This is due to the fact that the result of this multiplication must be overwritten where previously the array **b** was saved. If it is reached the penultimate polynomial

multiplication between arrays, the secret counter is reset, so that during the execution of the next polynomial multiplication the first secret polynomial in the secret key memory is loaded. In addition, the controller chooses the next octet of public coefficients increasing the counter of the memory from which it will be read. Meanwhile, the polynomial multiplier continues its execution, by consuming the second last public coefficient in public buffer, following the operations described in S5.

- **OCTET_RESMEM_PUBBUF (S18 state)** : in this state the controller continues the execution of the polynomial multiplication consuming the penultimate public coefficient left in the buffer. It also takes care of loading the new octet, read from the result key memory, in the upper part of the public buffer in order to guarantee continuity to the multiplication in the successive states. The controller also checks if 32 octets have already been read (256 coefficients in total), if this has not happened it returns to S5 to start consuming the coefficients of the new octet. It also checks whether the last octet of public coefficients of a array-array polynomial multiplication has been consumed, If this is the case, a new state will be created to use the last coefficient in the public buffer and then proceed with the rounding operation.

- **NEXT_OCTET_RES_MEM_SECR_LOAD_OP (S19 state)** : in this state the controller recognizes that will read the new octet of public coefficients from the result key memory, before doing this the result counter is increased so that it is read in the right location in the memory. The controller also checks the number of octets that have already been read. In case the FSM has read at least 16 of them and there are successive polynomial multiplications to be performed (for example, it is not reached the end of an array-array or matrix-array polynomial multiplication), the secret counter is incremented and the secret buffer is enabled. This will allow, in the next state, the reading of a block of 16 secret coefficients to be stored in the secret buffer. In subsequent uploads of the octets of public coefficients (16 times in total) will be read the blocks of the secret coefficients until it will be stored the next secret polynomial fully loaded in the secret buffer. Meanwhile, the polynomial multiplier continues its execution, by consuming the fourth last public coefficient in public buffer, following the operations described in S5. This state is only used during the execution of the array-array polynomial multiplication in both Encryption and Decryption functions.

- **NEXT_OCTET_RES_MEM_NO_OP (S20 state)** : in this state the controller recognizes that will read the new octet of public coefficients from the result key memory, before doing this the result counter is increased so that it is read in the right location in the memory. Meanwhile, the polynomial multiplier continues its execution, by consuming the fourth last public coefficient in public

buffer, following the operations described in S5. This state is only used during the execution of the array-array polynomial multiplication in both Encryption and Decryption functions.

- **ADD_K_KEYGEN (S21 state)** : in this state the controller starts the rounding operation used in the multiplication between matrix and vector. This operation is not only contain in the key generation function, but also in the encryption one. As the first step of rounding, the controller adds the polynomial constant to the result of the multiplication operation, selecting the corresponding input of the first multiplexer contained in the MACs.

- **SHIFT_MATRIX_ROUND (S22 state)** : in this state the controller terminates the rounding operation used in the multiplication between matrix and vector. This operation is not only present in the key generation function, but also in the encryption function. As the final step of rounding, the controller selects the right shift to apply to the result of the multiplication operation, picking the corresponding input of the second multiplexer contained in the MAC. The updated result is stored inside result buffer, At the end, the controller returns to the S5 state if the matrix-array multiplication is not yet finished.

- **SAVE_FINAL_RESULT (S23 state)** : in this state the controller starts saving the contents of the memory buffer in the result key memory, writing a block contained in the buffer and updating the result counter. This operation is repeated 32 times to ensure that the entire content of the result buffer is saved.

- **END_OPERATION (S24 state)** : in this state the controller raises the output signal finish to signal the end of the PKE function, using a specific version of Saber. All data calculated by the operation are saved in the result key memory.

- **FIRST_STEP_DECR_ROUND_LS (S25 state)** : in this state the controller starts the rounding operation of the PKE decryption function, using the LightSaber version of the protocol. As the first step of rounding, the controller subtracts, to the result of the multiplication operation, the correct left shift of each ciphertext coefficient, picking it from the first multiplexer contained in the MAC.

- **FIRST_STEP_DECR_ROUND_S (S26 state)** : in this state the controller starts the rounding operation of the PKE decryption function, using the Saber version of the protocol. As the first step of rounding, the controller subtracts, to the result of the multiplication operation, the correct left shift of each ciphertext coefficient, picking it from the first multiplexer contained in the MAC.

- **FIRST_STEP_DECR_ROUND_FS (S27 state)** : in this state the controller starts the rounding operation of the PKE decryption function, using the

FireSaber version of the protocol. As the first step of rounding, the controller subtracts, to the result of the multiplication operation, the correct left shift of each ciphertext coefficient, picking it from the first multiplexer contained in the MAC.

- **SECOND_STEP_DECR_ROUND (S28 state)** : in this state the controller continues the rounding operation of the PKE decryption function, for each version of Saber. As the next step of rounding, the controller adds, to the updated result of the multiplication operation, the polynomial constant by selecting it again from the first multiplexer contained in the MAC.

- **THIRD_STEP_DECR_ROUND (S29 state)** : in this state the controller completes the rounding operation of the PKE decryption function, for each version of Saber. As the final step of rounding, the controller selects the correct shift to the left of the updated result of the multiplication operation, selecting it from the second multiplexer contained in the MAC. It also loads in the result counter the memory address of the result key memory which will allow to write the rounded result in the right location. Finally, it is loaded into the result buffer.

- **FIRST_STEP_ENCR_ROUND (S30 state)** : in this state the controller starts the rounding operation of the PKE encryption function for each of Saber versions. As the first step of rounding, the controller adds, to the result of the multiplication operation, the correct left shift of each message coefficient, picking it from the first multiplexer contained in the MAC. Then, the controller choose next state depending of the Saber version used.

- **SECOND_STEP_ENCR_ROUND_LS (S31 state)** : in this state the controller terminates the rounding operation of encryption PKE function for the LightSaber version of the protocol. As the final step of rounding, the controller selects the right shift to apply to the updated result of the multiplication operation, picking the corresponding input of the second multiplexer contained in the MAC. The updated result is stored inside result buffer. The controller also loads in the result counter the memory address of the result key memory which will allow to write the rounded result in the right location. At the end, the controller returns to the S4 state to start the matrix-array multiplication contained in the function.

- **SECOND_STEP_ENCR_ROUND_LS (S32 state)** : in this state the controller terminates the rounding operation of encryption PKE function for the Saber version of the protocol. As the final step of rounding, the controller selects the right shift to apply to the updated result of the multiplication operation, picking the corresponding input of the second multiplexer contained in the MAC. The updated result is stored inside result buffer. The controller also loads in the

result counter the memory address of the result key memory which will allow to write the rounded result in the right location. At the end, the controller returns to the S4 state to start the matrix-array multiplication contained in the function.

- **SECOND_STEP_ENCR_ROUND_LS (S33 state)** : in this state the controller terminates the rounding operation of encryption PKE function for the FireSaber version of the protocol. As the final step of rounding, the controller selects the right shift to apply to the updated result of the multiplication operation, picking the corresponding input of the second multiplexer contained in the MAC. The updated result is stored inside result buffer. The controller also loads in the result counter the memory address of the result key memory which will allow to write the rounded result in the right location. At the end, the controller returns to the S4 state to start the matrix-array multiplication contained in the function.

- **END_SINGLE_PMULT_PUBMEM_8_RSTRESCNT (S34 state)** : in this state the controller loads the secret polynomial and the new octet of public coefficients. The first, contained in the memory buffer, is loaded inside the secret buffer while the last, which has been read from the public key memory, is loaded at the bottom of the public buffer of the polynomial multiplier. The controller recognizes that the current single polynomial multiplication concerned the last multiplication between polynomial arrays (specifically the first matrix-array multiplication), then the controller resets result counter of result key memory. This is due to the fact that, when the array-array multiplication will be end, the computed result will start to be loaded at the beginning of this memory. At the end, the controller will resume the execution by proceeding with the last the multiplication operation. Meanwhile, the polynomial multiplier continues its execution, by consuming the last public coefficient in public buffer, following the operations described in S5.

- **LOAD_OCTET_RESMEM_PUB_BUFF (S35 state)** : in this state the controller reads the first octet of public polynomials from the result key memory and loads it into the lowest part of the public buffer inside the polynomial multiplier while, in the secret buffer is loaded the secret polynomial, read in S2, contained in the memory buffer.

- **END_SINGLE_PMULT_RESMEM_8_RSTSECRCNT (S36 state)** : in this state the controller loads the secret polynomial and the new octet of public coefficients. The first, contained in the memory buffer, is loaded inside the secret buffer while the last, which has been read from the result key memory, is loaded at the bottom of the public buffer of the polynomial multiplier. The controller recognizes that the current single polynomial multiplication concerned the second last multiplication between polynomial arrays, then the controller resets secret

counter of secret key memory. This is due to the fact that, during the last polynomial multiplication involved in the arrays polynomial multiplication, the first secret polynomial will be loaded ready to be used during the next multiplication between arrays. At the end, the controller will apply the rounding operation during the next states. Meanwhile, the polynomial multiplier continues its execution, by consuming the last public coefficient in public buffer, following the operations described in S5.

- **END_SINGLE_POLYN_MULT_RESMEM_8 (S37 state)** : in this state the controller loads the secret polynomial and the new octet of public coefficients. The first, contained in the memory buffer, is loaded inside the secret buffer while the last, which has been read from the result key memory, is loaded at the bottom of the public buffer of the polynomial multiplier. If the current single polynomial multiplication concerned the last multiplication between polynomial arrays, then the controller will continue with the rounding operation in subsequent states, otherwise it will resume the execution of the multiplication operation. Meanwhile, the polynomial multiplier continues its execution, by consuming the last public coefficient in public buffer, following the operations described in S5.

- **SETUP_DECR_ROUND (S39 state)** : in this state the controller enables the memory buffer and the result counter, which will be used to store, in the next state, the contents of the buffer in the result key memory.