HIGH-SPEED IMPLEMENTATION OF POST-QUANTUM CRYPTOGRAPHY MULTIVARIATE SIGNATURE SCHEMES

by

Ahmed Ferozpuri
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Electrical and Computer Engineering

Committee:

_____ Dr. Kris Gaj, Thesis Director

_____ Dr. Jens-Peter Kaps, Committee Member

_____ Dr. Avesta Sasan, Committee Member

_____ Dr. Monson H. Hayes, Chairman,
Department of
Electrical and Computer Engineering

_____ Dr. Kenneth S. Ball, Dean,
Volgenau School of Engineering

Date: _____ Fall 2017
George Mason University
Fairfax, VA

High-Speed Implementation of Post-Quantum Cryptography Multivariate Signature Schemes

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science at George Mason University

By

Ahmed Ferozpuri
Bachelor of Science
George Mason University, 2007

Director: Dr. Kris Gaj, Associate Professor
Department of Electrical and Computer Engineering

Fall 2017
George Mason University
Fairfax, VA

Acknowledgments

I would like to thank my advisor Kris Gaj, who has not only helped me to accomplish this thesis, but has developed in me a strong interest in cryptography, number systems, digital design, and a belief in myself to persevere through failure. Also, I would like to acknowledge the infinite patience of my wife, Kiran Rizvi, who has supported me through out this difficult process. Finally, I don't think I have seen a kinder group of people than the Cryptography Engineering and Research Group (CERG), who have provided great ideas, feedback, and discussion in world-class cryptographic research.

Table of Contents

List of Tables

List of Figures

Abstract

HIGH-SPEED IMPLEMENTATION OF POST-QUANTUM CRYPTOGRAPHY MULTIVARIATE SIGNATURE SCHEMES

Ahmed Ferozpuri

George Mason University, 2017

Thesis Director: Dr. Kris Gaj

Multivariate cryptosystems belong to the five most promising families of post-quantum cryptography (PQC) schemes. Among them, the Unbalanced Oil and Vinegar (UOV) and the Rainbow signature schemes have been extensively studied since 1999 and 2005, respectively. The main advantage of UOV is high confidence in its security; the disadvantages include large key and signature sizes. Rainbow is a multi-layer version of UOV that offers better performance, smaller keys, and smaller signatures. In this thesis, we present and compare hardware implementations of both schemes in high-performance Field Programmable Gate Arrays (FPGAs). The optimization is for the minimum signature generation and verification time. The generation of keys is assumed to be done in software. Compared to the previous state-of-the-art high-speed implementation, the proposed design for Rainbow is more than twice as fast, and introduces two architectural innovations: a novel pivot calculation circuit and a memory based microprogrammed architecture. Additionally, in order to make benchmarking easier and fairer, our design follows a universal PQC hardware API, which allows for fair comparison with other post-quantum signature schemes, in particular those submitted to the NIST PQC Project. The design is intended to be made open-source

to speed-up further optimizations. Additionally, we will provide a projection of scalability for larger security levels and future optimizations.

Chapter 1: Introduction

In the mid-1990s Peter Shor published his ground-breaking paper detailing a quantum algorithm capable of solving factoring problems in polynomial time. The idea of a quantum computer may have seemed more like science fiction then, but recent developments indicate that we may be only a few decades away from a reliable and scalable quantum computer. In 2015, the United States National Security Agency (NSA) announced that those who have not yet switched over to Suite B recommendations, should hold off and instead wait for future post-quantum algorithms [1].

Is this really enough? Would a quantum computer armed with a sufficient number of qubits be capable of attacking larger key size versions of RSA, ECDSA, ECDH, and DSA? As pointed out in [2] and [3], only McEliece, NTRU, and Lattice-based schemes would not be broken by a large-scale quantum computer and AES and SHA with large key and output size respectively would be considered secure. For factoring based algorithms, it may be possible that a larger key size allows for some level of security until quantum systems advance far enough to support large-scale computation, and this may be what NSA was intending with their recommendation. However, public key systems requiring forward-secrecy would still be vulnerable because a large-scale quantum may be capable of obtaining keys for past data/key communications.

Those who require forward-secrecy must consider adopting a scheme now, which offers resistance to quantum and classical cryptanalysis in the near future. There are five major classes of post-quantum algorithms: lattice, code, hash, isogeny and multivariate-based. Practical post-quantum schemes can be used today to allow a forward-secure key exchange, such as, Ring-LWE, which is a lattice-based cryptosystem [JCMD 15]. Of course other possibilities exist, but require investigation and research and a comprehensive analysis of performance, security, and parameter set trade-offs.

In December 2016 the National Institute of Standards and Technology (NIST) made an announcement regarding their "Post-Quantum Cryptography Project", where they are accepting proposals for "quantum-resistant" public-key cryptographic algorithms [4]. There are many likely candidates based on hash functions, codes, lattices, elliptic curve isogenies, and multivariate quadratic systems, and as of this writing there have been 82 submissions with 2 withdrawals. Those that have withstood attacks for the longest are the most promising. Many experts agree that multivariate signature schemes offer a great potential to be used as quantum-resistant public key cryptosystems [2].

1.1 Multivariate-based Cryptography

Multivariate systems utilize a multivariate quadratic polynomial system to perform the cryptographic operations of a multivariate public key cryptosystem (MPKC). The earliest paper presenting a MPKC was that of Shigeo Tsujii and Hideki Imai in the early 1980s [2]. The system of multivariate polynomials is known as the central map, and is used as part of the private key for signature schemes. The structure of this central map is hidden by affine transformations, which are also included in the private key. During signature generation, inverting this map is required and is equivalent to solving a system of multivariate polynomials. The hardness of MPKC systems is based on the difficulty in solving multivariate polynomial system of equations over a finite field. Indeed, the key to a secure MPKC is the design of the central map and the corresponding trap door. Additionally, for efficiency, quadratic equations are used.

1.2 Early Multivariate-based Cryptography

The earliest standard (bipolar) construction seems to be relatively intact in modern multivariate cryptography. The public is key is made up of m polynomials with n variables, making an m-by-n matrix. Using two affine maps, S and T, the central map F can be hidden, which is an idea introduced in 1988 by Matsumoto and Imai. The central map is a class

of quadratic maps, whose inverse is easy to compute. The secret key consists of S, T, and information about F. It is obvious that this cryptosystem suffers from an issue with large public key sizes, but "formidable" private key sizes. The first multivariate cryptosystem used to generate a signature, which also initiated research into multivariate cryptosystems, used two quadratic variables and a prime modulus made up of large prime factors. It did not take long for this cryptosystem to be shown unsecure through cryptanalysis.

Pollard and Schnorr, showed that the first multivariate cryptosystem, could be broken without knowing the prime factors. It was shown that in fact, it is very difficult to build a practical system with two variables. Matsumoto, Imai, Harashima and Miyagawa, built a multivariate system with four variables and public keys are given by quadratic polynomials, but it was also broken shortly after by [5], and it was realized that more than four variables are required. The balanced oil-vinegar technique was introduced in [6] and broken shortly after by [7]. It was determined for the oil-vinegar method to work, we must select $o = 2v$, where o is the number of oil terms and v is the number of vinegar terms. Since then, UOV has remained unbroken. Rainbow was introduced in [8] and uses a layering technique to reduce key sizes. Similarly, Perturbed Matsumoto-Imai (PMI+) was developed to improve efficiency, where polynomials are perturbed and added [9]. However, these two schemes was defeated shortly after being proposed by [10] by using differential cryptanalysis, but Rainbow was reparameterized in [11] and since then has remained unbroken.

1.3 Security and Efficiency Improvements

Multivariate cryptography initially suffered from security problems, which were addressed through many improvements. Therefore, a necessary skepticism exists for currently proposed multivariate schemes, which are proposed recently, due to a lack of research. Parameter sets for higher security are available and methods to generate parameter sets are available in [12] and [13]. However, as pointed out by [14], some schemes such as ABC Encryption may have difficulty in scaling to larger security levels, and indeed the size of the

system solver becomes a limiting factor in high-speed implementations.

While other post-quantum categories discussed so far offer higher security levels, security reductions, and efficient implementations, one of the multivariate schemes biggest challenges is confidence in its security. Part of the difficulty is that it is more recently introduced into the cryptography community and awaits cryptanalysis. However, there are a number of attacks on multivariate schemes, most notably the MinRank attack, Grobner bases and the brute force attack with a gray-code representation for efficiency. In fact, the MQ Challenge website [15] shows many parameter sets for Multivariate schemes that have currently been broken for which many used Gray-code enumeration running on 128 Spartan 6 FPGAs. However, there is a threshold to what parameter sets can be attacked, and therefore, choosing a higher security level parameter set is sufficient.

One of the early efficiency improvements for multivariate schemes were triangular, oil and vinegar, and similar constructions that were directly vulnerable to rank attacks. These early constructions are known as single-field, and later big-field constructions were used that allow the central map to be hidden by two invertible affine linear maps. Big-field constructions require solving a system of equations in order to invert the central map, and this may be a time consuming process. One example of these is the Quartz algorithm, which was submitted to the New European Schemes for Signatures, Integrity and Encryption (NESSIE) and had the record for the slowest execution time for signing of half a minute. Improvements in speed have been demonstrated by HFEv- schemes, such as Gui, which reduce the signature time by a factor of 1000! The efficiency improvement was done by carefully choosing the parameters that affect the central map. In general, Internal Perturbation (IP) is known as using vinegar variables with projection, and prevents against differential attacks [9].

Another security improvement was the plus and minus method, where random polynomials were added or removed from the central map and denoted with a + or - symbol in the name of the algorithm. The minus method is limited to signature schemes, where a one-to-one mapping is not needed. This is due to the significant slowdown required to guess the missing coordinates. The plus method can be used in encryption, but causes a

slowdown for signature generation. The goal of Rainbow was to reduce the key size, which it does effectively. In general, there is much more confidence in Multivariate systems as a signature scheme, and therefore our focus will be on UOV and Rainbow, which is a layering scheme based on UOV.

Chapter 2: Previous Work

To fit different device constraints, some targeted hardware implementations for multivariate schemes have been implemented: [16], [17] and [18]. The goal in [17] is similar to our aim in this thesis - to optimize mainly for speed. The time-area optimized implementations [18] and [16] require more clock cycles but are optimal for smaller devices, such as those required in the Internet of Things (IoT). Alternatively, implementing a system solver capable of processing the entire matrix in parallel requires comparatively a lot of resources, but allows the most time savings in terms of clock cycles.

2.1 Systolic Arrays

The systolic array architecture for system solving was introduced in [19] as an array of processors, or processing elements, able to perform system solving using backward substitution. The physical structure is similar to an upper triangular matrix, and it uses two different PEs one type along the diagonal elements and the others to perform elimination. Using shift registers, the input matrix is inputted to the PE array, and operations are performed on the input and the results are stored in the PEs. Data can be moved between rows and columns, and all inputs/outputs of are registered. Therefore, the critical path is determined soley by the critical path of the largest PE. Their architecture requires a total of $3n$ clock cycles to solve a system and another n clock cycles to read out the solution, amounting to $4n$ total clock cycles to obtain the solution.

2.2 Systolic Network

In [20], the systolic network for system solving was described. The authors eliminate, the shift registers required for data input/output and registers in between PEs. This allows signals to propagate through the PE array in one clock cycle at the cost of a much larger critical path, which is directly related to the size of the matrix. Therefore, as n grows, this method becomes unfeasible. It takes $2n$ clock cycles to solve a system.

2.3 Systolic Line

Parallel systolic line architectures for system solvers in $GF(2)$ include SMITH [21], which solves a system using Gaussian Elimination with simultaneous backward substitution, and it offers a tradeoff between systolic arrays and networks. For a $n \times n$ system it takes on average $2n$ clock cycles and worst-case time complexity of $O(n^2)$. Balasubramanian et al. [16], [22] extend SMITH directly to solve matrices in multivariate schemes such as Rainbow, and call their new architecture G-SMITH. This architecture can solve a system with elements in $GF(2^t)$, in about $2n$ clock cycles without pipelining and about n clock cycles with pipelining. However, the pipelined version would not save clock cycles in UOV and Rainbow because only one system is solved followed by block multiplications.

2.4 Parallel Designs

Tang et al. [17] base their work on [19], [20] and [21], which can solve a system in $4n$, $2n$, $2n$ (average) clock cycles, respectively, and they were the first to provide a hardware architecture that solves a system in n clock cycles, after the first pivot is calculated. We provide a similar parallel architecture requiring a greatly reduced number of multiplexers and with the same time complexity of $n + 1$ clock cycles. Additionally, it includes key improvements that give a significant speed up for signature generation and verification along with a more regular structure for the system solver.

2.5 Our Contribution

2.5.1 Speedup

The goal of this thesis is to focus on speeding up the execution time of UOV and Rainbow, without consideration to area. Compared to previous work by Tang et al., this implementation completes signature generation and verification more than twice as fast. This speedup is enabled by hardware support for the reuse of all multipliers in the system solver (through multiplexing) and a reduction in the total number of multiplications required for signature generation and verification, which will be elaborated in Section 6.3.3. This allows for simultaneously increasing the number of multiplications per clock cycle and reducing the total multiplications required. These enhancements significantly effect the signature generation and verification time. This is evident from observing timing results from [17] that show the total percentage of clock cycles spent on solving a system becomes only (12.12%) of total execution time. We effectively reduce the time for the other 87.88% of execution time using the same number of processing elements.

2.5.2 Novel Pivot Circuit

The benefits of a swapless pivot in this high-speed architecture will be described briefly in comparison to the Tang et al. architecture. First, many multiplexers are eliminated, which are required for the Tang et al. design and indicated in the functional description as the internal "sending" of rows and elements to elimination, normalization, and inverse components after the matrix is "sent" to the system solver. This removes the strong dependence of the architecture to the matrix size, since there is an exponential growth in the size these multiplexers with larger systems. Obviously with a swapless design, the multiplexers required for swapping positions are eliminated.

Next, multiplexers required to perform elimination and normalization on the correct rows, which required choosing between using either the current or next row, are eliminated by utilizing a processing array that gives the system solver a more regular structure. This

array is made up of uniform processing elements (PE) that can perform both normalization and elimination on any row and is configured by the pivot circuit, which effectively eliminates the need to internally "send" rows or elements within the system solver. Lastly, multiplexers required to obtain the current pivot row and column are eliminated by instead generating indexes that an external circuit uses, such as memory, to provide the pivot row and column as inputs to the system solver. Similar to the Tang et al. design, we can generate the pivot for the next iteration, but also require registering the PE array's operation. The operation of the pivot circuit is described in Section 5.

2.5.3 Memory-based microprogramming

The benefit of using a memory-based microprogramming approach is the ease of scheduling operations. It is used in this architecture based on temporary values stored, such as the selection of a pivot from the matrix or the current line of the public or private key being processed. In our architecture, we apply memory-based microprogramming in both the system solver and the scheduling of inputs for block multiplication. In the system solver, it will be shown how memory is used to instruct the PE array to operate on the input data. The scheduling of inputs for block multiplication uses circular shift registers that contain the proper input sequences required for the current line of memory. This approach is also extensible for variable width multiplier arrays, however, we target the maximum re-use of multipliers in this thesis.

2.5.4 Matrix Binarization

We introduce a new concept called Matrix Binarization (MB), that can convert a system with elements in $GF(2^t)$ in to an equivalent binary system with elements in $GF(2)$. MB can be extended to the *Affine Transformation*, *Polynomial Evaluation* and *System Solving* steps, and can be applied to the keys in a form of preprocessing or performed on the fly. This allows operations required for signature generation and verification to be performed using only AND/XOR gates. Also, there is no need for inversion when *System Solving* is

performed. There are modifications required to the PEs, to support only AND/XOR gates with multiplexers, and to PivotCalc to control the new PE array. The resulting circuit will have a much smaller critical path and would require less area. *System Solving* would require t more clock cycles, but there can be a drastic decrease in the number of clock cycles required for *Affine Transformation*, *Polynomial Evaluation*, which will be discussed in Section 9.2. This is important since more than 70% of execution time is spent on these operations, which can make up for the increase in clock cycles required for *System Solving*.

Chapter 3: Multivariate Cryptography

3.1 Background, Notation, and Operations

Multivariate cryptography uses a system of non-linear multivariate polynomials to perform signature generation, encryption, and key exchange. In multivariate cryptography, a set of polynomials, $\mathcal{P}$, represents the public key and is made up of m polynomials in n variables as shown in (3.1). Additionally, the degree of the system, d, is 2, in order to constrain the key size and the amount of computations.

$$
\begin{aligned}
p^{(1)}(x_1, ..., x_n) &= \sum_{i=1}^{n} \sum_{j=i}^{n} p_{ij}^{(1)} x_i x_j + \sum_{i=1}^{n} p_i^{(1)} x_i + p_0^{(1)} \\
p^{(2)}(x_1, ..., x_n) &= \sum_{i=1}^{n} \sum_{j=i}^{n} p_{ij}^{(2)} x_i x_j + \sum_{i=1}^{n} p_i^{(2)} x_i + p_0^{(2)} \\
&\cdots \\
p^{(m)}(x_1, ..., x_n) &= \sum_{i=1}^{n} \sum_{j=i}^{n} p_{ij}^{(m)} x_i x_j + \sum_{i=1}^{n} p_i^{(m)} x_i + p_0^{(m)}
\end{aligned}
\tag{3.1}
$$

For all multivariate public key cryptosystems, the size of the public key is based on the number of terms in the polynomial, which depends directly on m, n, and d. Using (3.1), there are $\binom{n}{2} + n = \binom{n+1}{2}$ $p_{ij}^{(k)}$ terms, n $p_i^{(k)}$ terms and 1 constant for any given kth polynomial in $\mathcal{P}$. Therefore, the size of the public key, s, for m polynomials, is shown in (3.2).

$$s = m\left(\binom{n+1}{2} + n + 1\right) = m\binom{n+2}{2} \tag{3.2}$$

As shown in (3.1) the basic operations are addition and multiplication of elements $p_{ij}^{(k)}, p_i^{(k)}, x_i, x_j$, which are elements of finite field $\mathbb{F}_q$, where we choose the parameter $q = p^t = 2^8$. The choice of $p = 2$ allows for addition to be only an XOR operation. Multiplication between field elements is done using polynomial multiplication and more detail is given in section 4. Additionally, the system of multivariate equations must be solved, and system solvers are covered in section 5. In order to generate the public and private key pairs, invertible affine transformations must be determined, however, this operation is assumed to occur in software as specified in [23], and the hardware receives inverted affine transformations as part of the private key input.

3.2 Unbalanced Oil and Vinegar (UOV)

The original Oil and Vinegar scheme [6] is described below.

Let $\mathbb{F}_q$ be the finite field and o and v be integers such that $n = o+v$. Let $\mathcal{S} = 1, ..., v$ and $\mathcal{O} = v+1, ..., n$ be sets of integers representing the indices of variables. Let $\mathbf{S}$ be the set of variables, $(x_i \mid i \in \mathcal{S})$ called Vinegar variables. Let $\mathbf{O}$ be the set of variables $(x_i \mid i \in \mathcal{O})$, and are the Oil variables. Let $\mathbf{x} = x_1,\ x_2,\ \ldots\ x_n$, where each $x_i \in \mathbb{F}_q$. The central map, $\mathcal{F}$, where $\mathcal{F} : \mathbb{F}^n \to \mathbb{F}^o$, contains o polynomials in n variables, $p^{(1)},\ p^{(2)},\ \ldots, p^{(o)} \in \mathbb{F}[\mathbf{x}]$ in the form;

$$p^{(k)}(\mathbf{x}) = \sum_{i,j \in V, i \leq j} \alpha_{ij}^{(k)} x_i x_j + \sum_{i \in V, j \in O} \beta_{ij}^{(k)} x_i x_j + \sum_{i \in V \cup O} \gamma_i^{(k)} x_i + \eta^{(k)}, \tag{3.3}$$

where k is the number of a specific polynomial in the map.

Public Key: In order to hide the composition of the central map, one affine transformation, $\mathcal{T}$, where $\mathcal{T} : \mathbb{F}^n \to \mathbb{F}^n$, is composed with $\mathcal{F}$. Therefore, the public key is defined as $\mathcal{P} = \mathcal{F} \circ \mathcal{T}$ where $\mathcal{P} : \mathbb{F}^n \to \mathbb{F}^o$. Since the composition in a standard bi-polar construction does not change the structure of $\mathcal{F}$ it is not needed for the security of UOV [13].

Private Key: The private key is made up of $\mathcal{F}$ and $\mathcal{T}^{-1}$ and therefore the size is $|\mathcal{F}| + |\mathcal{T}|$, which contains randomly generated coefficients, where $\mathcal{T}$ is ensured to be invertible. Let $f = |\mathcal{F}|$ and $t_1 = |\mathcal{T}|$.

Using (3.3), the types of the coefficients in a given polynomial will be further identified to derive the size of the $\mathcal{F}$. Let terms corresponding to coefficients α_{ij} be denoted as vinegar-vinegar (VV). Let terms corresponding to coefficients β_{ij} be denoted as vinegar-oil (VO). Let terms corresponding to coefficients γ_i be denoted as vinegar-oil-only (VOO). Let the term corresponding to the coefficient η be denoted as a constant (C). The number of coefficients of each type is given by; $|VV| = \binom{v+1}{2}$, $|VO| = v * o$, $|VOO| = v + o = n$, $|C| = 1$

For each polynomial, $p^{(k)}$;

$$|p^{(k)}| = |VV| + |VO| + |VOO| + |C| = \binom{v+1}{2} + v * o + n + 1 \tag{3.4}$$

Since $\mathcal{F}$ contains o polynomials, $|\mathcal{F}| = o * |p^{(k)}|$.

Since $\mathcal{T}$ is an affine transformation it is made up of a matrix $\mathbf{M_T} \in \mathbb{F}^{n \times n}$ and a vector $\mathbf{c_T} \in \mathbb{F}^n$, $|\mathcal{T}| = n \cdot (n+1)$

Therefore, the size of the private key is:

$$o \cdot \left(\binom{v+1}{2} + v * o + n + 1 \right) + n \cdot (n+1) \tag{3.5}$$

Signature generation: Prior to generating the signature, a hash, **msg_in**, of the message, **msg**, is generated, such that, $\mathcal{H} : \{0,1\}^\star \rightarrow \mathbb{F}^o$ and $\mathbf{msg\_in} = \mathcal{H}(\mathbf{msg}) \in \mathbb{F}^o$. In order to calculate the signature, **sgn_out**, the following calculations using the private key are performed: $\mathbf{sgn\_out} = (\mathcal{T}^{-1} \circ \mathcal{F}^{-1})(\mathbf{msg\_in})$. The first step is to find $\mathbf{O} = \mathcal{F}^{-1}(\mathbf{msg\_in})$. In order to do this, the Vinegar variables, **S**, must be chosen at random. *Polynomial Evaluation* is used to reduce the quadratic polynomials into a linear system of equations, called $\mathcal{F}'$, which can be solved by the *System Solver* to obtain the set of Oil variables, **O**. The system may fail to be solved with low probability and the process will have to be repeated. Next, matrix multiplication and vector addition is performed in *Affine Transformation* to find the signature $\mathbf{sgn\_out} = \mathcal{T}^{-1}(\mathbf{O})$. This process is illustrated in Fig. 3.1. Note: Bus widths for diagrams in the Background section are a multiple of the parameter $t = 8$, and bolded arrows represent multi-bit signal buses.

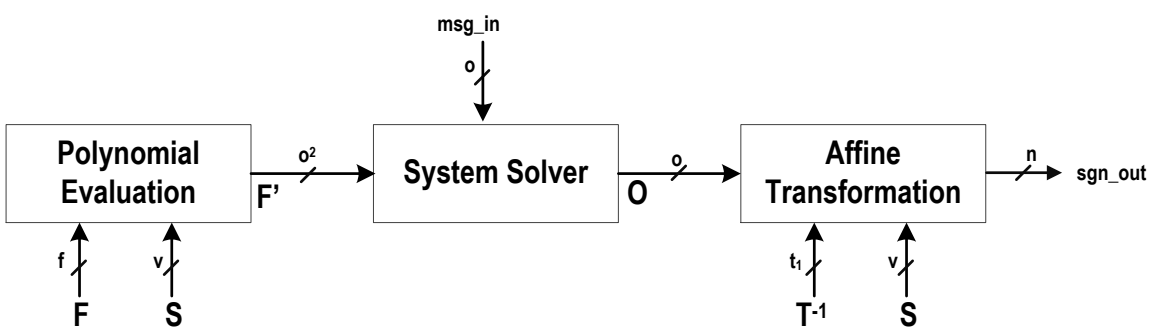

Figure 3.1: UOV Signature Flow Diagram

Signature verification: Prior to verifying a signature is valid, a hash is computed such that $\mathbf{msg\_in} = \mathcal{H}(\mathbf{msg})$. Next the signature, **sgn_in**, is used to produce $\mathbf{msg\_out} = \mathcal{P}(\mathbf{sgn\_in})$ through *Polynomial Evaluation*. If $\mathbf{msg\_in} = \mathbf{msg\_out}$ the signature is valid and the *is_valid* signal is set to 1, otherwise 0. This process is illustrated in Fig. 3.2. Note, for UOV $m = o$.

In the original scheme, $v = o$, but it was proposed in [24] to set $v = 2o$ to defend against new attacks.

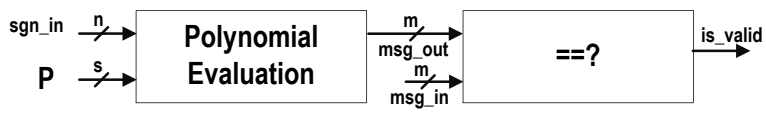

Figure 3.2: UOV and Rainbow Verification Flow Diagram

3.3 Rainbow

The original Rainbow scheme [8] is very similar to UOV, but introduces a layering technique to reduce the signature, public and private key sizes.

Let $\mathbb{F}_q$ be a finite field and $\mathcal{S}$ be the set $\{1,\ 2,\ \ldots, n\}$. Let $v_1,\ v_2\ \ldots, v_n$ be integers such that $0 < v_1 < v_2\ \ldots\ < v_u < v_{u+1} = n$. Define sets $\mathcal{S}_i = \{1,\ 2,\ \ldots, v_i\}$ for $i = \{1,\ 2,\ \ldots, u\}$. Let $o_i = v_{i+1} - v_i$ and $\mathcal{O}_i = \{v_i + 1,\ v_i + 2,\ \ldots, v_{i+1}\}$ for $i \in \mathcal{U}$ such that $\mathcal{U} = \{1,\ 2,\ \ldots, u\}$. Therefore, $|\mathcal{S}_i| = v_i$ and $|\mathcal{O}_i| = o_i$. Let $\mathbf{x} = x_1,\ x_2,\ \ldots\ x_n$, where each $x_i \in \mathbb{F}_q$.

The central map, $\mathcal{F} : \mathbb{F}^n \to \mathbb{F}^m$ and contains $m = n - v_1$ polynomials, $p^{(1)},\ p^{(2)},\ \ldots, p^{(m)}\ \in \mathbb{F}[\mathbf{x}]$ in the form;

$$p^{(k)}(\mathbf{x}) = \sum_{i,j \in S_\ell, i \leq j} \alpha_{ij}^{(k)} x_i x_j + \sum_{i \in O_\ell, j \in S_\ell} \beta_{ij}^{(k)} x_i x_j + \sum_{i \in S_\ell \cup O_\ell} \gamma_i^{(k)} x_i + \eta^{(k)}, \tag{3.6}$$

where k is a specific polynomial in the map and ℓ is the only integer such that $k \in \mathcal{O}_\ell$.

Let $\mathbf{S}_\ell$ be the set of variables $(x_i \mid i \in \mathcal{S}_\ell)$ called the Vinegar variables for layer ℓ. Let $\mathbf{O}_\ell$ be the set of variables $(x_i \mid i \in O_\ell)$ called the Oil variables for layer ℓ. Additionally, let $\mathbf{S}$ and $\mathbf{O}$ be the set of all vinegar and oil variables, respectively. If $u = 1$, the central map is the same as the UOV central map described in 3.2.

Public Key: In order to hide the composition of the central map, two affine transformations are composed with $\mathcal{F}$, $\mathcal{L}_1 : \mathbb{F}^m \to \mathbb{F}^m$ and $\mathcal{L}_2 : \mathbb{F}^n \to \mathbb{F}^n$. Therefore, the public key is defined as $\mathcal{P} = \mathcal{L}_1 \circ \mathcal{F} \circ \mathcal{L}_2$ where $\mathcal{P} : \mathbb{F}^n \to \mathbb{F}^m$.

Private Key: The private key is made up of $\mathcal{F}$, $\mathcal{L}_1^{-1}$ and $\mathcal{L}_2^{-1}$ and therefore the size is $|\mathcal{F}| + |\mathcal{L}_1| + |\mathcal{L}_2|$, which contains randomly generated coefficients, where $\mathcal{L}_1$ and $\mathcal{L}_2$ are ensured to be invertible. Let $\mathcal{F}_\ell$ be the set of polynomials for a given layer, and $f_\ell = |\mathcal{F}_\ell|$. Let $l1 = |\mathcal{L}_1|$ and $l2 = |\mathcal{L}_2|$.

Since $\mathcal{L}_1$ is an affine transformation it is made up of a matrix $\mathbf{M_{L_1}} \in \mathbb{F}^{m\times m}$ and a vector $\mathbf{c_{L_1}} \in \mathbb{F}^m$, $|\mathcal{L}_1| = m \cdot (m+1)$ and since $\mathcal{L}_2$ is an affine transformation it is made up of a matrix $\mathbf{M_{L_2}} \in \mathbb{F}^{n\times n}$ and a vector $\mathbf{c_{L_2}} \in \mathbb{F}^n$, $|\mathcal{L}_2| = n \cdot (n+1)$.

Using the UOV private key size shown in equation (3.5) and extending it to the Rainbow layering scheme, the size of the private key is:

$$\sum_{l=1}^{u} o_l \cdot \left(\binom{v_l+1}{2} + v_l * o_l + v_{l+1} + 1 \right) + m \cdot (m+1) + n \cdot (n+1) \tag{3.7}$$

Signature generation: Prior to generating the signature, a hash, **msg_in**, of the message, **msg**, is generated, such that, $\mathcal{H} : \{0,1\}^\star \to \mathbb{F}^m$ and $\mathbf{msg\_in} = \mathcal{H}(\mathbf{msg}) \in \mathbb{F}^m$. In order to calculate the signature, **sgn_out**, the following calculations using the using the private key are performed: $\mathbf{sgn\_out} = (\mathcal{L}_2^{-1} \circ \mathcal{F}^{-1} \circ \mathcal{L}_1^{-1})(\mathbf{msg\_in})$. For simplicity, assume $u = 2$, and therefore $\ell \in (1,2)$. The first step is to apply a matrix multiplication and vector addition using *Affine Transformation* to find $\mathbf{Y} = \mathcal{L}_1^{-1}(\mathbf{msg\_in})$, where $\mathbf{Y} = \mathbf{Y_1}|\mathbf{Y_2}$, such that $\mathbf{Y}_l \in \mathcal{F}^{o_l}$. Next, the central map must be inverted through a recursive process that produces each layer's Oil set $\mathbf{O}_\ell = \mathcal{F}_\ell^{-1}(\mathbf{Y}_\ell)$. *Polynomial Evaluation* is used to reduce the quadratic polynomials into a linear system of equations, called $\mathcal{F}'_\ell$, which can be solved by the *System Solver* to obtain $\mathbf{O}_\ell$. This process continues until we obtain the last set of Oil variables $\mathbf{O}_u$. Next, matrix multiplication and vector addition is performed in *Affine Transformation* to find the signature $\mathbf{sgn\_out} = \mathcal{L}_2^{-1}(\mathbf{S}|\mathbf{O})$. In order to avoid re-doing all steps if an inconsistent system is detected, vinegar variables can optionally be chosen at each layer, and this is denoted in Fig. 3.3 as a dashed arrow for the set of vinegar variables

chosen in the second layer, called S_2'.

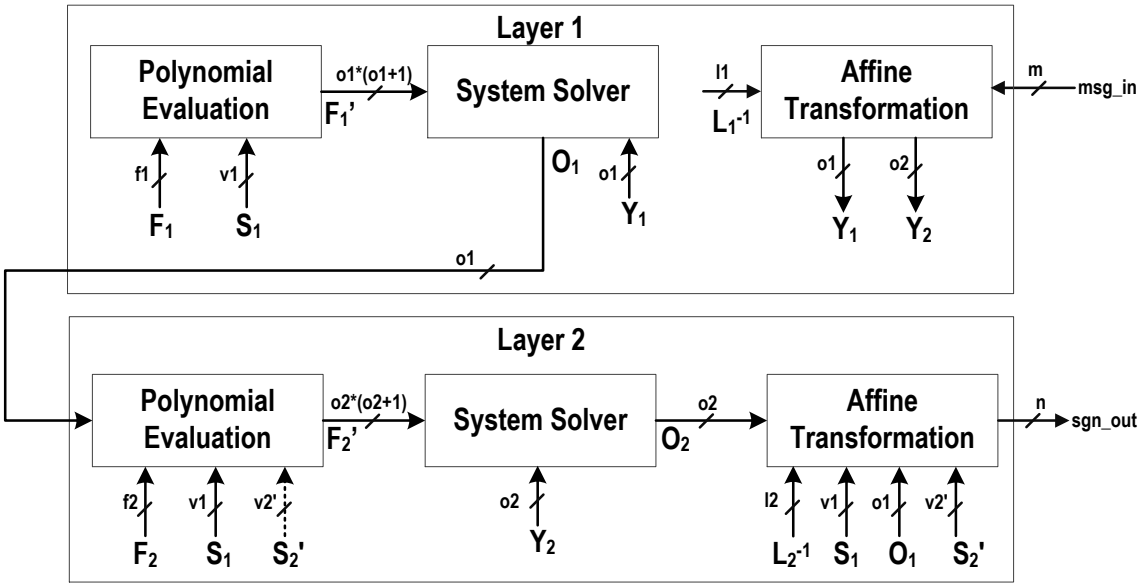

Figure 3.3: Rainbow Signature Flow Diagram

Signature verification: The process of Rainbow signature verification is identical to the one described above for UOV, shown in Fig. 3.2.

3.4 Parameter Sets

For each scheme $\mathbb{F}_q = GF(2^8) = GF(256)$. The parameters for each scheme are shown in Table 3.1

Table 3.1: UOV Parameter Selection

Sec level (bits)	***(o, v)***	***Pub Key size (kB)***	***Priv Key size (kB)***	***Hash size (bits)***	***Sign size (bits)***
80	(28, 56)	99.9	95.8	224	672
128	(45, 90)	409.4	381.8	360	1,080

All parameter sets taken from [13].

Table 3.2: Rainbow Parameter Selection

Sec level (bits)	***(v_1,o_1),(v_2',o_2)/ (v_1,o_1,o_2)***	***Pub Key size (kB)***	***Priv Key size (kB)***	***Hash size (bits)***	***Sign size (bits)***
80[11]	(17,12),(1,12)*	22.17	17.06	192	336
128[12]	(28, 20, 20)	94.3	62.9	320	544

*For comparison with [17].
Note: $v2'$ indicates the number of vinegar variables chosen in the second layer, which is equivalent to S_2' in Fig. 3.3

Chapter 4: Polynomial Multipliers

The basic operations in multivariate cryptography include multiplication and addition over a finite field, q. Additionally, multiplications of elements in q are reduced using a reduction polynomial. Similar to [17], and for efficiency reasons, the reduction polynomial used in this thesis is $x^8 + x^6 + x^3 + x^2 + 1$. Several state-of-the-art multipliers were created, verified in VHDL and will be compared at the end of this section.

4.1 Mastrovito Multiplier

Zhang et al describe a method to create Mastrovito multipliers using general irreducible polynomials in [25]. There are two methods proposed, while the latter one, called "Modified Mastrovito Multiplier" works better with high-Hamming weight reduction polynomials. The first "Mastrovito Multiplier" method is used in this thesis. Their paper provides a theorem and method to construct a product matrix $\mathbf{M}$. Next, an algorithm is presented to compute $\mathbf{M}$ using a series of Toeplitz matrices, where subexpression sharing is utilized to create a highly modular architecture that works for any irreducible polynomial.

The general algorithm produces a weighted tree, D, based on the irreducible reduction polynomial. The weights of nodes in the tree D are used to create a multiset, $\mathcal{H}$, which can contain duplicates. Next, each value $j \in \mathcal{H}$ is added to seperate multisets, S_j. If $|S_j|\ mod\ 2 = 1$, then j is inserted in to $\mathcal{N}$. The set $\mathcal{N}$ is the final result of this algorithm and it is used to produce inputs for an XOR Array in the Product Matrix Module. This is the only modular component in the design, which changes based on the irreducible polynomial. Upon specification of the XOR Array an appropriate circuit for polynomial multiplication can be constructed.

Results are given for both a two and three-input Mastrovito multiplier, where a couple of two-input Mastrovito multipliers were connected together to produce a single three-input multiplier.

4.2 Multiply-then-Reduce Multiplier

Tang et al describe a novel three-input multiplier to speed up Rainbow in [17]. The main idea is that in $GF(2^8)$ it is "faster" to multiply the three inputs first, then perform reduction using the irreducible polynomial, and as such this type of multiplier will be referred to as Multiply-then-Reduce (MTR). They claim that this method is "anti-intuitive" and only works for small fields, such as $GF(2^8)$ and is not applicable for large fields. However, the algorithm is quite straight forward, and Python was used to produce the corresponding sum-of-products form for both a two-input and three-input multiplier. For clarity, the algorithm is defined briefly here below;

Let $a(x) = \sum_{i=0}^{7} a_i x^i$, $b(x) = \sum_{i=0}^{7} b_i x^i$, and $c(x) = \sum_{i=0}^{7} c_i x^i$ such that $a(x),\ b(x),\ and\ c(x) \in GF(2^8)$ and in the standard polynomial basis. Let $d(x) = (a(x) \times b(x) \times c(x))(mod(f(x))) = \sum_{i=0}^{7} d^i x^i$, where f(x) is the irreducible polynomial.

1. Compute v_{ij} for $i = 0, 1, \ldots, 21$ and $j = 0, 1, \ldots, 7$ such that $x^i\ mod\ f(x) = \sum_{j=0}^{7} v_{ij} x^j$

2. Compute S_i for $i = 0, 1, \ldots, 21$ such that $S_i = \sum_{j+k+l=i}^{7} a_j b_k c_l$

3. Compute d_i for $i = 0, 1, \ldots, 7$ such that $d_i = \sum_{j=0}^{21} v_{ij} S_j$

4. Obtain the result $d(x) = \sum_{i=0}^{7} d_i x^j$

The sum of products expression in Step 3 is obtained for output bits of $d(x)$ in order to create a circuit for the corresponding three-input multiplier. Additionally, the same process was used to create a two-iput multiplier for comparison shown below in Table 4.1. Note that MTR-2 is an MTR multiplier with 2 inputs, MTR-3 is an MTR multiplier with 3 inputs and MTR-2x2 is two MTR-2 multipliers connected in series to produce a 3-input multiplier.

4.3 Multiplier Results

Table 4.1: $GF(2^8)$ Polynomial Multiplier Comparison

Name	***Number of Operands***	***Area (LUTs)***	***Max Frequency (MHz)***
Mastrovito	2	34	444.44
MTR-2	2	38	500.00
Mastrovito	**3**	**81**	**235.29**
MTR-2x2	3	90	250.00
MTR-3	3	312	235.29
MTR-? [17]	?	37*	92.87

*This value is given in ALUTs, for the Altera Stratix II family.

Chapter 5: System Solvers

5.1 Our Approach

In order to solve a system of equations, the Gauss-Jordan elimination method is used, which requires three basic operations: inversion, normalization and elimination. In order to solve a system in o_ℓ clock cycles, pivoting plus the basic operations must be completed each clock cycle. Additionally, to speedup the multiplication process for affine transformation and polynomial evaluation, all multipliers are re-used and this requires allowing a dual-mode operation at the cost of multiplexers and an additional input port denoted aux_in. Note, adders in LSS are not reused.

5.2 Interface and Operation

The interface diagram for the linear system solver (LSS) component is shown in Fig. 5.1, and bus widths are expressed in bits. The mat_in and aux_in have a width of $2 \times o_\ell \times (o_\ell + 1) \times t$ bits, and the thinner buses are single bit inputs. Data signals $mat1\_in$, $mat2\_in$, $aux1\_in$, and $aux2\_in$ are used in subsequent diagrams and are equal sized parts of mat_in and aux_in signals. These signals are used to represent inputs for solving a system or performing a batch of multiplications. Multiplication output data signals are $mat1\_out$ and $mat2\_out$, jointly represented as mat_out. The data signals are assumed to be a row vector form of a matrix consisting of elements $data(i, j)$ $\forall i, j \in \mathcal{O}_\ell$, where $data$ is any of the aforementioned data signal names. Additionally, pi, pj, pi_seq and $valid$ will be discussed in Section 5.2.2, and $p = \lceil log_2(o_\ell) \rceil$. Additionally, when p is used in figures, it will represent bits.

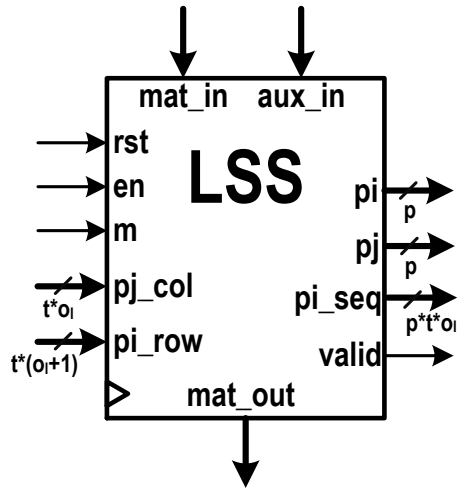

Figure 5.1: LSS interface diagram
Note: $p = \lceil log_2(o_\ell) \rceil$

5.2.1 Multiplication operation

When input $m = 1$, LSS is in the batch multiplication mode, where all data inputs are used. By utilizing all multipliers in the PEs, there are $2 \times o_\ell \times (o_\ell + 1)$ multiplications possible per clock cycle. For each clock cycle the result is available on the data output bus mat_out corresponding to the element-wise multiplication of $mat1\_in \times aux1\_in$ and $mat2\_in \times aux2\_in$, respectively.

5.2.2 System solving operation

When input $m = 0$, LSS is in the system solving mode, where only $mat1\_in$ is required to have valid data for a system that needs to be solved. Data from $mat1\_in$ is used as input to the PEs and the appropriate operations are performed based on the op. Upon every rising edge of the clk input after determining the first pivot, new iterations of the Gauss-Jordan elimination are produced, until the o_ℓ clock cycle - where the solution vector is available in the last column of $mat2\_out$. The detailed operation is shown in Algorithm 1.

Although this algorithm is shown sequentially, it is executed in parallel as illustrated in Fig. **??**. The first step is to set default values for the pivot column index, pj, and the pivot

Algorithm 1 Gauss-Jordan Elimination using *LSS*

1: **INPUT:** $mat1\_in$, pj_col, pi_row
2: **OUTPUT:** $mat2\_out$, pi, pj, pi_seq, $valid$ ▷ Outputs available per iteration
3: $pj \leftarrow 0$ ▷ Assign pivot column
4: $pi \leftarrow 0$ ▷ Assign pivot row (may change)
5:
6: **while** $pj < o_\ell + 1$ **do** ▷ The first iteration produces initial pivot
7: $(pi, pj, op, pi\_seq, valid) \leftarrow$ pivotCalc(pj_col) ▷ Execute pivotCalc
8: $\beta \leftarrow pj\_col(pi)$ ▷ β is the current pivot
9: $\beta^{-1} \leftarrow \mathrm{I}(\beta)$ ▷ Use inverse component to calculate β^{-1}
10: $mat2\_out \leftarrow \mathrm{PE\_ARRAY}(op, mat1\_in, pj\_row, pi\_col, \beta^{-1})$
11: **end while**
12:
13: **return** $(pi\_seq, mat2\_out(:, o_\ell))$ ▷ Last column in $mat2\_out$ is the solution

row index, pi. Then a while loop is run for $o_\ell + 1$ iterations. The first iteration is used to generate the signals shown in line 7. The op signal sets the PEs into the correct mode of operation to perform either elimination or normalization. The pi based on the current pivot is calculated and pi_seq has the current pi shifted in. However, pi_seq is not useful until the system is solved, and is used to reorder the solution column to obtain an ordered solution set. Lastly, the $valid$ bit indicates if the system is inconsistent.

Next, the current pivot, β, is selected using multiplexers and the previously registered value of p_i, which is not illustrated. The inverse, β^{-1} is computed using the inverse component, I. On line 10, the PE_ARRAY function call indicates a block normalization and elimination operation on the $mat1\_in$ by the PE array using the specified inputs. Normalization and Elimination are defined by operations on elements found in the inputs: $mat1\_in$, pi_row, pj_col and β^{-1}. Let $PE_{i,j}$ indicate a processing element at a the ith row and jth column in the PE array. Then elimination is $mat2\_out(i,j) = pi\_row(j) \times pj\_col(i) \times \beta^{-1} + mat1\_out(i,j)$, which is equivalent to taking the appropriate multiple of the corresponding element in the pivot row and normalizing it using β^{-1} and then adding. Normalization is $mat2\_out(i,j) = mat1\_in(i,j) \times \beta^{-1}$, which is equivalent to multiplying the current element by the inverse of the pivot. It is worth noting that all PEs use β^{-1} to eliminate the dependency of elimination operations on normalization, which

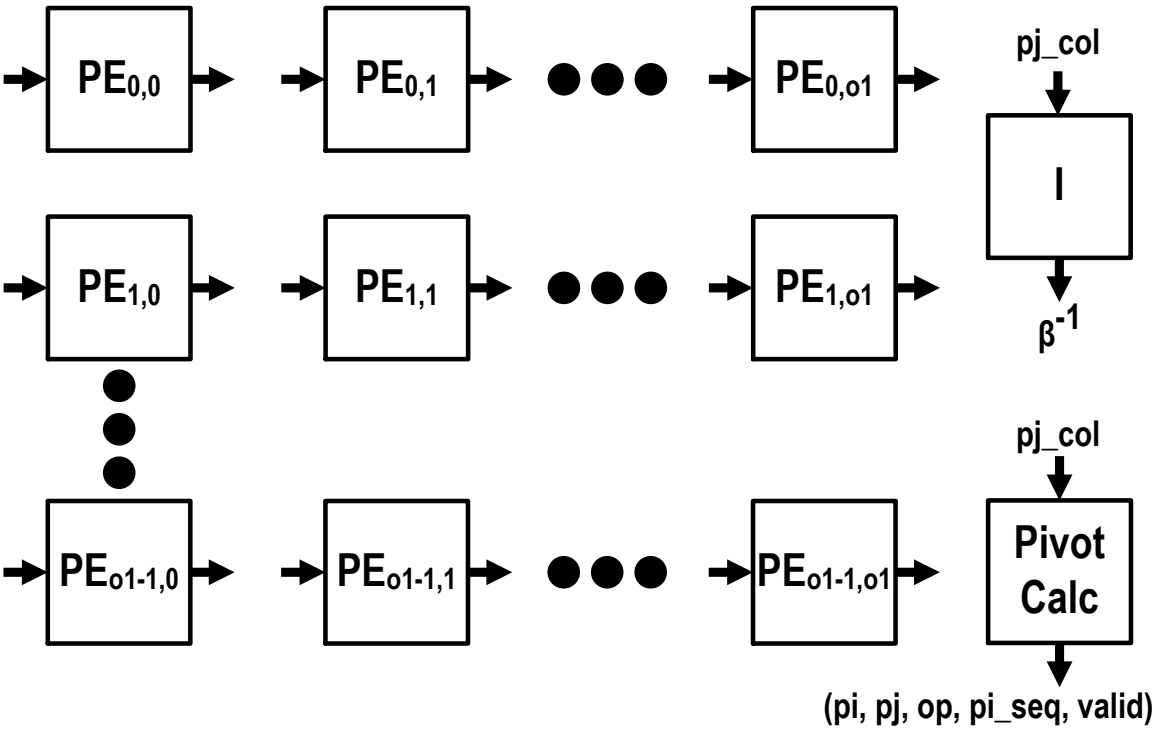

Figure 5.2: LSS Top-level Diagram

eliminates n clock cycles.

Finally, pj is incremented in order to process the next column in the system and the loop continues until iteration $o_\ell + 1$, where the solution is available in the last column of $mat2\_out$ denoted, $mat2\_out(:, o_\ell)$. Additionally, an ordered solution can be obtained using pi_seq.

5.2.3 Operational Example

This section gives an example of solving a 5×5 system with elements in $GF(2^8)$ using LSS. The detailed steps for Gauss-Jordan elimination on a matrix is shown in Table 5.1. For each iteration, important rows and columns are bolded. The first iteration is required to determine the first pivot, and the only part of the matrix used is the pivot column. The "X" in $mat2\_out$ indicates a don't care because the result is not used. Iterations 2-6 bold the pivot row and pivot column based on the pj and pi indexes generated by PivotCalc.

Additionally, each step also shows the internal values of PivotCalc that are used to control the PE array. The final result is shown on in the last column of $mat2\_out$ on iteration 6.

5.2.4 Pivot Circuit

A pivot component that keeps track of previous pivot locations and can seamlessly compute new pivots was created. The diagram for the pivot calculation component called, $PivotCalc$, is shown in Fig. 5.3. It takes as input, each element in the current pivot column, pj_col. Each element is input into o_ℓ comparators, denoted by a "$! = 0$" block with a corresponding enable, en_i. If the en_i signal is 1, then the output of the comparator will always be 0, and otherwise, it will function normally. Based on the output of the comparators, an o_ℓ bit signal, $neq0_i$, is sent to a priority encoder. The priority encoder will output the pivot row index, pi.

In order to generate en_i, the output pi is sent to a decoder to produce a local signal dec_out, which is registered to produce op that programs the PE array for the next clock cycle. Also, the decoder is necessary because $neq0_i$ may have more than one new potential pivot, and therefore, op will have a 1 at only the newly selected pivot position, which is exactly the same row of elements that needs to perform normalization. Next, dec_out is XOR'd with previously stored values of en_i and the result will be stored in the en_i register. Each clock cycle, the calculated pi is stored in a SIPO and produces pi_seq once the system is solved. Additionally, the $valid$ flag indicates if the system is inconsistent, and allows the controller to break out of the current system solving operation immediately. It only checks if all bits in $neq0_i$ are all set to 0, and if so, the system is inconsistent.

5.2.5 Inversion

This design uses a partial multiplicative inverse (PMI) based on Fermat's theorem as described in [17]. Fig. 5.4 shows the block diagram of the partial multiplicative inverse component (PMI), denoted as (I), and otherwise noted the bus width is t bits. The input to this component are all the possible elements in the pivot column, denoted as pi_col and

Table 5.1: Steps for solving a system using *LSS*

it#	**mat1_in**	**mat2_out**	***Signals***
1	$\begin{bmatrix} \mathbf{1b} & 43 & d6 & 42 & 41 & e6 \\ \mathbf{13} & 4d & 5e & a9 & 17 & 81 \\ \mathbf{12} & 83 & 79 & 02 & 17 & df \\ \mathbf{f3} & 67 & f2 & 5d & 67 & f5 \\ \mathbf{1e} & 98 & f6 & 39 & 97 & ef \end{bmatrix}$	X	pj=0 , pi=0
2	$\begin{bmatrix} \mathbf{1b} & \mathbf{43} & \mathbf{d6} & \mathbf{42} & \mathbf{41} & \mathbf{e6} \\ \mathbf{13} & 4d & 5e & a9 & 17 & 81 \\ \mathbf{12} & 83 & 79 & 02 & 17 & df \\ \mathbf{f3} & 67 & f2 & 5d & 67 & f5 \\ \mathbf{1e} & 98 & f6 & 39 & 97 & ef \end{bmatrix}$	$\begin{bmatrix} 01 & f4 & 4b & 2b & 07 & 11 \\ 00 & 00 & 4a & fe & 6e & ef \\ 00 & 3a & 26 & 7e & 69 & a0 \\ 00 & 82 & 61 & eb & 24 & 6a \\ 00 & 28 & 0a & fc & cd & 5c \end{bmatrix}$	pj=1 , pi=2 neq0= $\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix}$ op= $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}$ eni= $\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \end{bmatrix}$
3	$\begin{bmatrix} 01 & \mathbf{f4} & 4b & 2b & 07 & 11 \\ 00 & \mathbf{00} & 4a & fe & 6e & ef \\ \mathbf{00} & \mathbf{3a} & \mathbf{26} & \mathbf{7e} & \mathbf{69} & \mathbf{a0} \\ 00 & \mathbf{82} & 61 & eb & 24 & 6a \\ 00 & \mathbf{28} & 0a & fc & cd & 5c \end{bmatrix}$	$\begin{bmatrix} 01 & 00 & 69 & cf & d8 & 78 \\ 00 & 00 & 4a & fe & 6e & ef \\ 00 & 01 & a9 & 6e & a4 & 23 \\ 00 & 00 & 10 & 2f & 2c & 48 \\ 00 & 00 & cb & 5d & 89 & 10 \end{bmatrix}$	pj=2 , pi=1 neq0= $\begin{bmatrix} 0 & 0 & 1 & 1 & 1 \end{bmatrix}$ op= $\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix}$ eni= $\begin{bmatrix} 0 & 1 & 0 & 1 & 1 \end{bmatrix}$
4	$\begin{bmatrix} 01 & 00 & \mathbf{69} & cf & d8 & 78 \\ \mathbf{00} & \mathbf{00} & \mathbf{4a} & \mathbf{fe} & \mathbf{6e} & \mathbf{ef} \\ 00 & 01 & \mathbf{a9} & 6e & a4 & 23 \\ 00 & 00 & \mathbf{10} & 2f & 2c & 48 \\ 00 & 00 & \mathbf{cb} & 5d & 89 & 10 \end{bmatrix}$	$\begin{bmatrix} 01 & 00 & 00 & ef & e5 & 75 \\ 00 & 00 & 01 & 52 & d5 & ae \\ 00 & 01 & 00 & be & 92 & ad \\ 00 & 00 & 00 & 3b & ba & c0 \\ 00 & 00 & 00 & 51 & fb & 68 \end{bmatrix}$	pj=3 , pi=3 neq0= $\begin{bmatrix} 0 & 1 & 0 & 1 & 1 \end{bmatrix}$ op= $\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix}$ eni= $\begin{bmatrix} 0 & 0 & 0 & 1 & 1 \end{bmatrix}$
5	$\begin{bmatrix} 01 & 00 & 00 & \mathbf{ef} & e5 & 75 \\ 00 & 00 & 01 & \mathbf{52} & d5 & ae \\ 00 & 01 & 00 & \mathbf{be} & 92 & ad \\ \mathbf{00} & \mathbf{00} & \mathbf{00} & \mathbf{3b} & \mathbf{ba} & \mathbf{c0} \\ 00 & 00 & 00 & \mathbf{51} & fb & 68 \end{bmatrix}$	$\begin{bmatrix} 01 & 00 & 00 & 00 & b4 & 54 \\ 00 & 00 & 01 & 00 & 9f & 4e \\ 00 & 01 & 00 & 00 & aa & 9a \\ 00 & 00 & 00 & 01 & da & 52 \\ 00 & 00 & 00 & 00 & 92 & 7e \end{bmatrix}$	pj=4 , pi=4 neq0= $\begin{bmatrix} 0 & 0 & 0 & 1 & 1 \end{bmatrix}$ op= $\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix}$ eni= $\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \end{bmatrix}$
6	$\begin{bmatrix} 01 & 00 & 00 & 00 & \mathbf{b4} & 54 \\ 00 & 00 & 01 & 00 & \mathbf{9f} & 4e \\ 00 & 01 & 00 & 00 & \mathbf{aa} & 9a \\ 00 & 00 & 00 & 01 & \mathbf{da} & 52 \\ \mathbf{00} & \mathbf{00} & \mathbf{00} & \mathbf{00} & \mathbf{92} & \mathbf{7e} \end{bmatrix}$	$\begin{bmatrix} 01 & 00 & 00 & 00 & 00 & \mathbf{4f} \\ 00 & 00 & 01 & 00 & 00 & \mathbf{00} \\ 00 & 01 & 00 & 00 & 00 & \mathbf{25} \\ 00 & 00 & 00 & 01 & 00 & \mathbf{22} \\ 00 & 00 & 00 & 00 & 01 & \mathbf{31} \end{bmatrix}$	pj=X , pi=X neq0= $\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \end{bmatrix}$ op= $\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \end{bmatrix}$ eni= $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ pi_seq= $\begin{bmatrix} 0 & 2 & 1 & 3 & 4 \end{bmatrix}$

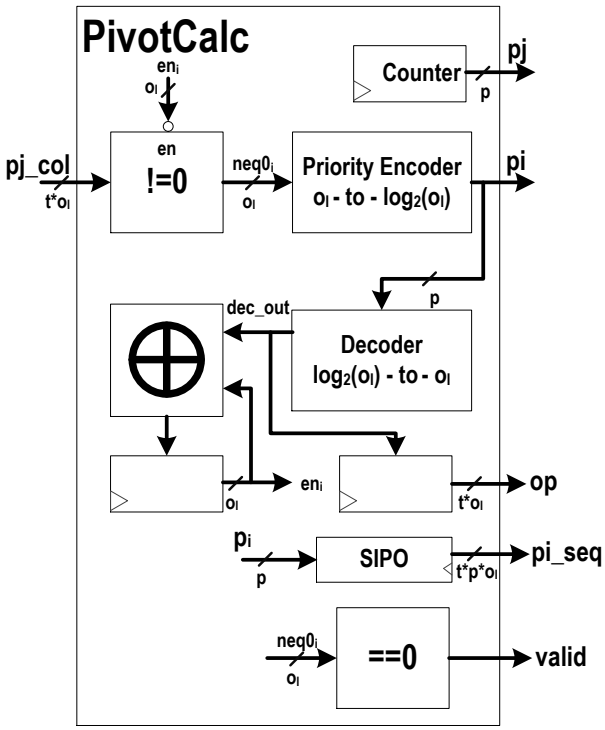

Figure 5.3: PivotCalc Component

using the pivot row index, p_i, to choose the appropriate pivot element. In order to compute the partial multiplicative inverse, each component $\beta^2, \beta^4, \ldots, \beta^{128}$ must be calculated. The *PMICalc* component produces the values of (β^2, β^4, β^8) and (β^{16}, β^{32}, β^{64}), which are sent to two three-input multipliers to produce S_1 and S_2. The triple (S_1, S_2, β^{128}) are used in the PE components and are equivalent to β^{-1}.

5.2.6 PEs for Elimination, Normalization and Multiplication

The processing element shown in Fig. 5.5 is capable of producing both the elimination and normalization of any element in the matrix. There are in total, 8 multiplexers, which are required for batch multiplication mode. If the select is not labeled, it is only the signal m. Fig. 5.5 depicts a positionally fixed processing element, which has a row value of i and a column value of j. These indices are used to connect appropriate elements from inputs with subscript i and/or j. When $m = 1$ and $op = 1$, the component performs

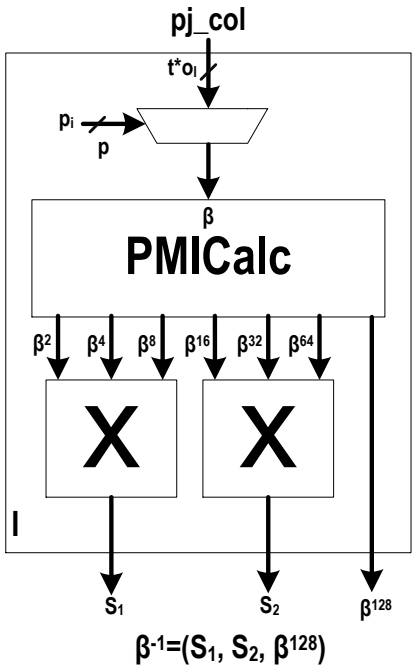

Figure 5.4: Inverse Component

$mat1\_out = mat1\_in \times aux1\_in$ and $mat2\_out = mat2\_in \times aux2\_in$.

When $m = 0$, the PE is in system solving mode and performs either normalization or elimination when $op = 0$ and $op = 1$, respectively. In both cases, the result is available on $mat2\_out_{i,j}$, where i and j represent a position in the matrix. When $op = 0$, the inputs are the corresponding matrix element, denoted $mat1\_in_{ij}$ and β^{-1}, which are used by the PE to produce the normalized result. When $op = 1$, the inputs are the corresponding element in the pivot column, denoted pj_col_i, the corresponding element in the pivot row, denoted pi_row_j, and β^{-1}. The result of this multiplication is added to $mat1\_in_{ij}$ to produce the result of elimination.

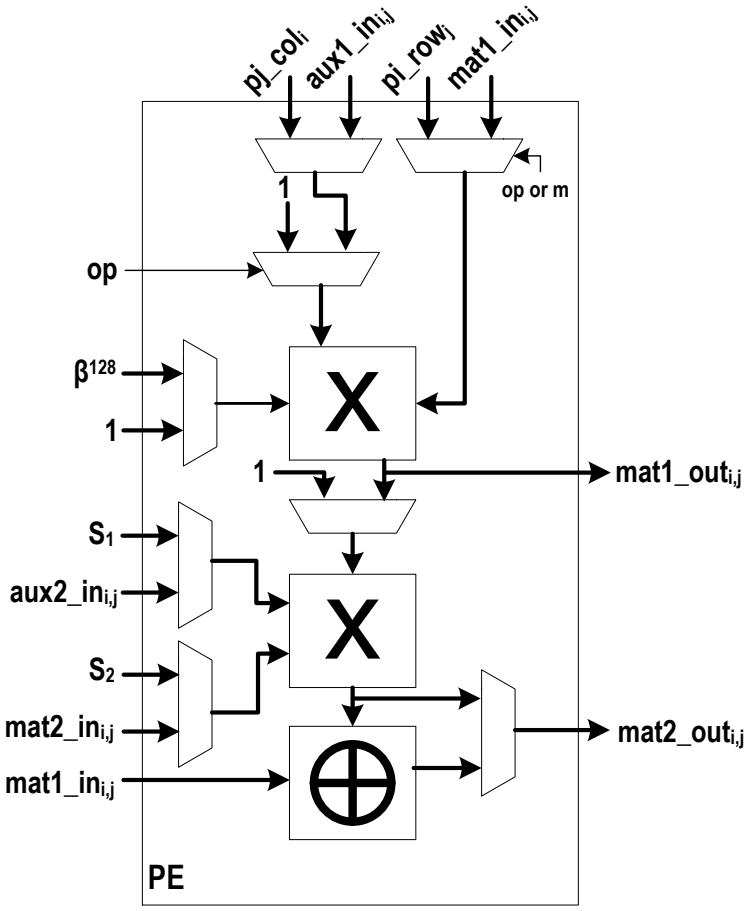

Figure 5.5: Processing Element

Chapter 6: High-Speed Implementations

Table 6.1: Rainbow (17, 12)(1, 12) Signature Steps

Step	***Desc***	***Operation***	***Mults***	***Cycles***	***Mults****	***Cycles****
1	AT	$msg\_in + c_1 = a_1$	N/A	1	N/A	1
2	AT	$L_1^{-1} \times a_1 = Y = Y_1 \| Y_2$	576	2	576	4
3	PE	$F_1(S_1) = F_1'$	4641	16	6324	45
4	SS	Solve $F_1' = Y_1$	N/A	13	N/A	12
5	PE	$F_2(S_1 \| S_2' \| O_1) = F_2'$	10725	35	15840	111
6	SS	Solve $F_2' = Y_2$	N/A	13	N/A	12
7	AT	$(S_1 \| S_2' \| O_1 \| O_2) + c_2 = a_2$	N/A	1	N/A	1
8	AT	$L_2^{-1} \times a_2 = sgn\_out$	1764	6	1764	12
			Total	87		198

Let AT = *Affine Transformation*, PE = *Polynomial Evaluation*
SS = *System Solving*, Desc = Description,
Mults = Total Multiplications, Cycles = Total Clock Cycles
*Taken from [17]

The high-speed implementation capable of using the *LSS* component described above must be able to provide data inputs as required to perform each step of the UOV and Rainbow algorithm. For brevity and comparison to [17], only the Rainbow (17, 12), (1, 12) scheme's signature generation will be discussed in detail. However, a similar top-level implementation is used for both schemes and the process for verification is only a *Polynomial Evaluation* followed by a comparison, as shown in Fig. 3.2. The operations and timing of Rainbow, as shown in Fig. 3.3, are detailed in Table 6.1, which gives the number of multiplications and clock cycles required to execute each operation of signature generation. The total clock cycles required is 87, compared to 198 by [17].

LSS can of solve an $o_\ell \times o_\ell$ system in o_ℓ clock cycles after the first pivot is determined,

and since it uses all multipliers it can compute $2 \times o_\ell \times (o_\ell + 1)$ multiplications per clock cycle. For the chosen parameter set of Rainbow, $o_\ell = 12$, therefore, it takes 13 clock cycles for system solving, and it can compute 312 multiplications per clock cycle, which is double compared to Tang et al. The other speedup occurs by registering the result of VV multiplications that are reused in each polynomial. Total, there is a significant reduction in the number of multiplications required for polynomial evaluation and the number of clock cycles compared to [17].

6.1 Architecture Overview and Operation

All steps described above will be executed on the architecture shown in Figure 6.1. The Figure is split in to parts for the Preprocessor, Rainbow Core, Postprocessor, and external RAM. The the middle section represents the Rainbow core area that will be used for subsequent area calculations. The processes of executing *Affine Transformation*, *Polynomial Evaluation* and *System Solving* used in Table 6.1 are described below. This section will describe a functional overview of the top-level operation, but excludes error handling for inconsistent systems, which is handled by the controller.

Affine Transformation: The first AT is performed by reading lines of memory from $SIPORamLBS$ corresponding to L_1^{-1} and producing inputs to LSS that are made up of data from a_1, which is provided by $SeqGen$. Once LSS performs multiplication, the corresponding result is sent to $RowSum$ to produce $Y1$ and $Y2$, which contain a value for each polynomial in L_1^{-1}. Similarly, in step 8, from $SIPORamLBS$ corresponding to L_2^{-1} are multiplied with values from $SeqGen$, that are combined by $RowSum$ to produce the sgn_out.

Polynomial Evaluation: The first PE is performed by reading lines of memory from $SIPORamLBS$ corresponding to each polynomial in F_1 and producing inputs to LSS that

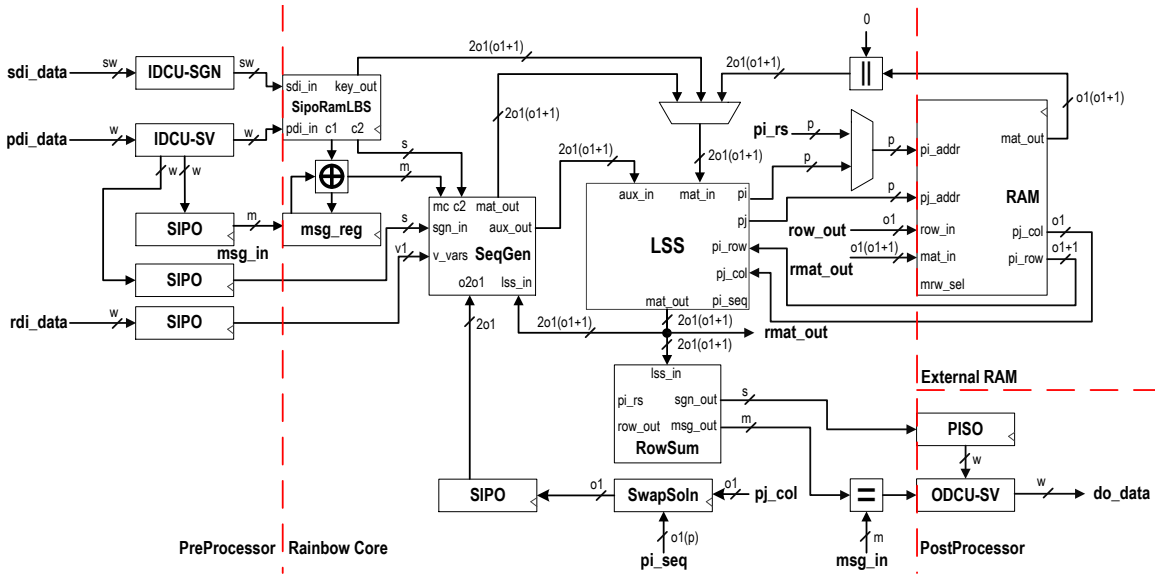

Figure 6.1: Rainbow Top-Level Diagram

are made up of data from S_1, which is provided by $SeqGen$. Once LSS performs multiplication, the corresponding result is sent to $RowSum$ to produce the reduced form of each polynomial in F_1 made up of oil variable coefficients and a constant term, which is sent to memory. In this fashion, the entire matrix, F_1' is produced. Similarly, in step 5, lines from $SIPRamLBS$ corresponding to F_2 are multiplied with values from from $SeqGen$, which are combined by $RowSum$ to produce F_2'.

System Solving: Prior to running the system solver to solve $F_1^{-1} = Y_1$, the constant terms in the system are summed with Y_1 or Y_2 and input to $mat1\_in$ of LSS from $RowSum$. The system solver will provide the solution vector, O_ℓ, in $o_\ell + 1$ clock cycles and the result will be stored in RAM. The solution vector is then reordered using the $SwapSoln$ component.

6.2 Sequencing via Memory-based Microprogramming

The process of applying *Affine Transformation* and *Polynomial Evaluation* requires data from memory representing the public and private keys. For this reason, when the system is executing AT and PE, it does so by operating on one line of memory at time from $SIPORamLBS$. The pattern operation executed on the architecture depends on the specific line. For example, the operations in AT will require the multiplication of known variables and their coefficients from memory followed by an XOR of fixed groups in the result. These groupings depend on the particular AT and are equal in size to a row in the corresponding matrix. The pattern of inputs matching each coefficient are specified by $SeqGen$. The sequences are generated by a circular shift register that is loaded with an initial pattern that represents all variables required, in order, for a row of the matrix being operated on. The subsequent polynomial evaluation is performed on the result of LSS by $RowSum$, which also sends the result in to memory.

6.3 Components

In order to provide data for all operations in Table 6.1 and maintain compliance with [23], the top-level design shown in Fig. 6.1 was used. This section provides an overview of components that are used to accomplish signature generation and verification.

6.3.1 Protocol Processing

Loading data: The rdi_data, pdi_data and sdi_data ports read in data at $w = sw = 32$ bits and can operate independently. The smaller block width causes large load times as shwon in Table 8.1. The pdi_data port is connected to the input data conversion unit used in both signature generation and verification (IDCU-SV), and this component parses the protocol defined in [23]. The sdi_data port is connected to the input data conversion unit for only signature generation (IDCU-SGN), and is used to parse the secret key. The rdi_data port is used to read the random vinegar variables used for signature generation, which are stored

in a SIPO. The public data port, *pdi_data*, is used to read *msg_in* for signature generation or the public key for verification. The *msg_in* value is stored in a SIPO and sent to the *msg_in* register, *msg_reg*. Once $c1$ is available, it is XOR'd with *msg_reg* and available for *SeqGen* on the *mc* input port.

Outputting Signature / Verification: After the *sgn_out* is ready for transmission, it is stored in a PISO, and each block is sent out across *do_data* after passing through the output data conversion unit for signature generation and verification (ODCU-SV). This component will produce the appropriate segments for compliance with [23], and transmit *sgn_out*. Similarly, for verification, the PASS or FAIL segment is generated to indicate the status of verification.

6.3.2 SipoRamLBS

Both the public and private key are generated in software, as required by [23] and stored in the $SIPORamLBS$ component, which reads in w-bit blocks of data and stores it in a SIPO. Once the SIPO is full, it loads the data to an internal RAM component. The RAM is incrementally populated with a line of the the public and private key that is $2 \times o_\ell \times (o_\ell + 1)$ bytes wide, which is used in signature verification or generation, respectively. Additionally, when in read mode, the $SIPORamLBS$ also outputs an associated code for the memory location being read using a look-up table. This code is a crucial part of the memory-based microprogrammed architecture used to generate sequences and is connected to *SeqGen* and *RowSum*. All subsequent operations will operate on a line of memory at a time.

The top-level diagram for $SIPORamLBS$ is shown in Fig. 6.2. Lines of the key will be written to $RAM1$ for the public key and $RAM2$ for the private key depending on the *mode* signal. The memory height of the private key is (*mhpv*) and the memory height of the public key is *mhpb*. These values are used to initialize a decrementer, *mem_decr*, which is keeps track of the next memory location to write to. Since w is chosen as 32 a write to RAM occurs only when a word counter, *block_incr*, is equal to *mwdw*, setting *load_word* to

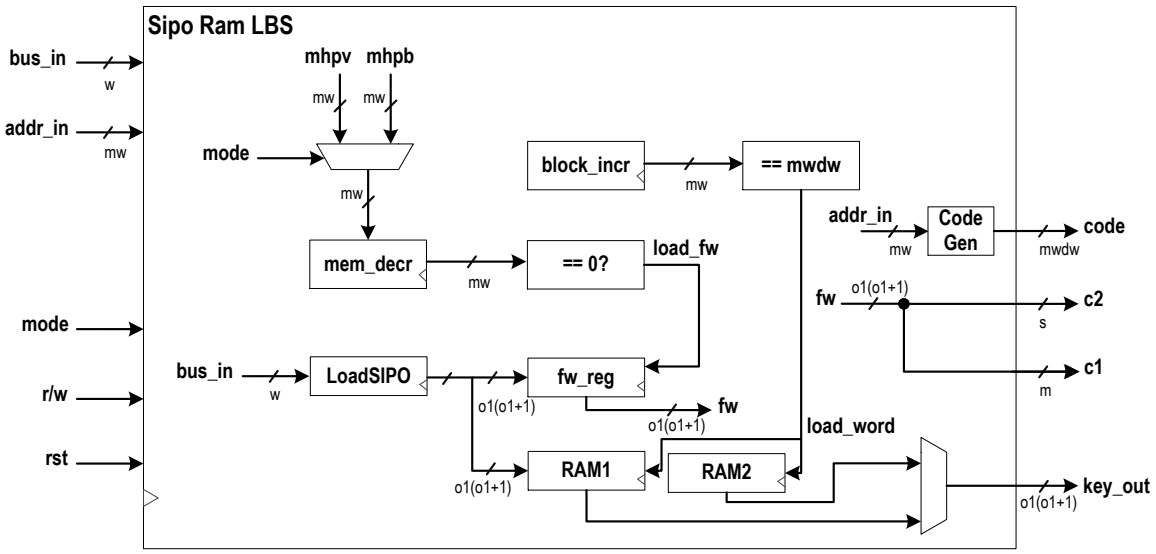

Figure 6.2: $SIPORamLBS$ Top-Level Diagram

true, and writing the contents of $LoadSIPO$ to either $RAM1$ or $RAM2$. Once mem_decr reaches 0, the fw_reg is used to load the last blocks of memory corresponding to $c1$ and $c2$. Once the private or public key is read, $c1$ and $c2$ are available for use. Then $SipoRamLBS$ is intended to be used in the read mode, where $addr\_in$ is used to read a line of the public or private key from key_out and the corresponding memory-code is output on the $code$ signal.

6.3.3 SeqGen

The sequence generator circuit ($SeqGen$) is able to provide the appropriate inputs across the aux_in bus that correspond with coefficients contained in the line of memory being processed from $SIPORamLBS$ for the private or public key based operations. It provides precise multiplicands corresponding to each byte in the current line of memory. For example, during the affine transformations, it will provide the correct sequence of bytes required to perform the matrix-vector multiplication. In Table 6.1, the matrices are L_1^{-1} and L_2^{-1} multiplied by the vectors are a_1 and a_2, respectively. Additionally, to perform Step 2 in Table 6.1, would require 2 clock cycles that correspond to processing two lines of memory. The data inputs to LSS are summarized in Table 6.2, and for brevity, the matrix L_1^{-1} is

shown as A, and the parenthesis represent elements in side the matrix or vector. The signal aux_in is generated by $SeqGen$ and mat_in is produced by $SIPORamLBS$.

Table 6.2: Affine Transformation steps

it#	**aux_in**	**mat_in**
1	[$a_1(0)$ $a_1(1)$ ... $a_1(23)$ $a_1(0)$ ... $a_1(23)$]	[$A(0)$ $A(1)$ ... $A(23)$ $A(24)$ ... $A(311)$]
2	[$a_1(0)$ $a_1(1)$ ... $a_1(23)$ $a_1(0)$ ... $a_1(23)$]	[$A(312)$ $A(313)$ ... $A(335)$ $A(336)$ ... $A(575)$]

The sequences required for aux_in are generated using a circular shift register that rotates by the multiplier width, $2 \times o_\ell \times (o_\ell + 1)$, each clock cycle. Similar to AT, matching inputs for each line of memory is used in polynomial evaluation. However, there is a SIPO available in SeqGen that stores the VV multiplications, which are reused for subsequent polynomial evaluations - effectively eliminating the need to perform these multiplications with each polynomial as done in [17].

The top-level diagram for $SeqGen$ is shown in Fig. 6.3. The outputs are mat_out and aux_out, which are inputs to LSS required to process the corresponding parts of the key. The input $code$ is used to choose the appropriate sequence for loading, seq_load, and shifting seq_shift, which are used to load the initial sequence based on the data inputs mc, v_vars, $c2$, $o2o1$, vv_vars and $sign\_in$, and shift sequences by the number of multipliers, $2 \times o_\ell \times (o_\ell + 1)$. The vv_vars signal is generated by first queuing the vinegar variables, v_vars for multiplication in LSS. On the next clock cycle, the next sequence is queued and the lss_in signal is is used to store the VV variables iteratively in to vv_sipo, and this allows for a large reduction in multiplications by instead reusing the result in vv_sipo in subsequent polynomials. The size of vv_sipo must support the maximum number of VV variables possible, which is s. The patterns $seqVV0$, $seqL1$, $seqPK$, etc are generated from components as shown in Fig. 6.3, where PK stands for public key. Additionally, the logic for these functions were generated by a parameterizable Python scripts.

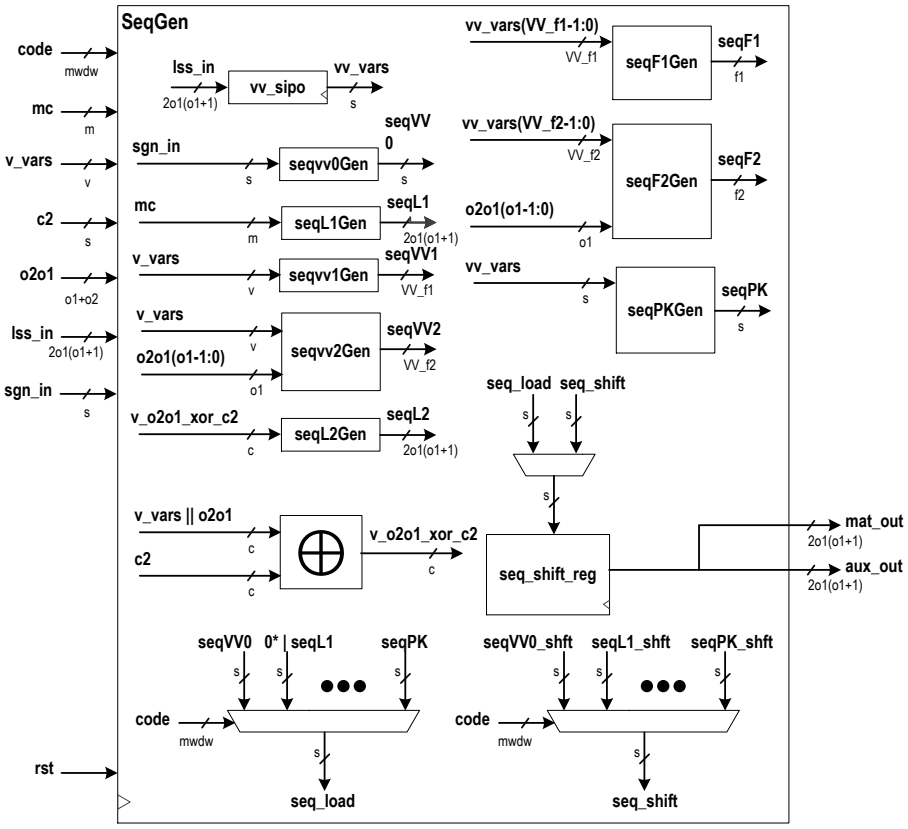

Figure 6.3: *SeqGen* Top-Level Diagram

6.3.4 RowSum

All outputs from LSS are sent to RowSum for bulk elimination. For AT, all lines of memory are stored in a SIPO. At most this will be 6 lines of memory for L_2^{-1}. The affine transformations produce $Y = Y1|Y2$, which is stored in RowSum, the signature sgn and the verification message ver. For PE, only one polynomial is reduced at time, and the matrix is updated accordingly (per row).

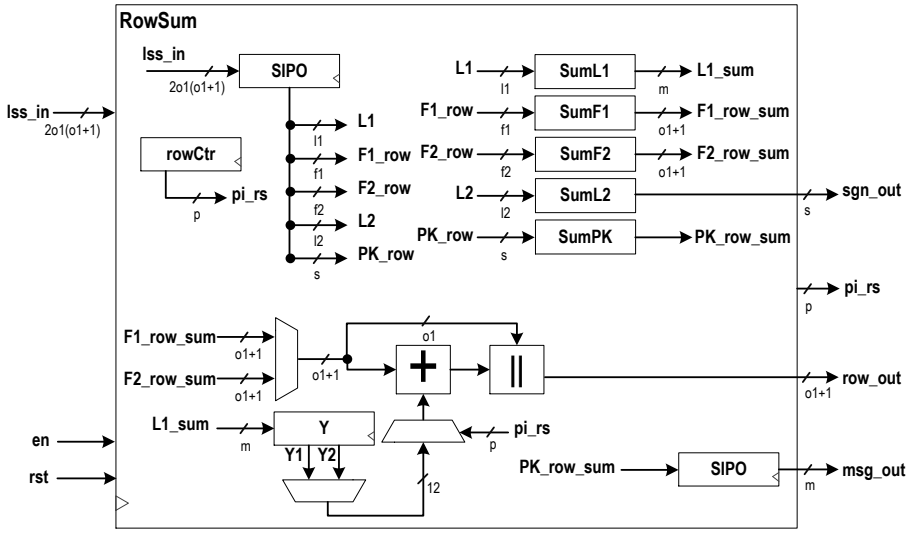

Figure 6.4: $RowSum$ Top-Level Diagram

The top-level diagram for $RowSum$ is shown in Fig. 6.4. The input lss_in is used to read the output of LSS of size $2 \times o_\ell \times (o_\ell + 1)$. These results are stored in a $SIPO$, which is connect to the summing logic, producing $L1\_sum$, $F1\_row\_sum$, $F2\_row\_sum$, sgn and PK_row_sum. Note, the summation logic is generated by parameterizable Python scripts. These sums are used to produce the results of summing the entire L1 or L2 or a row corresponding to a polynomial in F1, F2 or PK. In the case of PK, each summation results in a byte that is stored in SIPO used to generate msg_out. The summation of L1 is stored in the Y register, which has a separate output for $Y1$ and $Y2$. The incremental constant values of row updates for F1 and F2 are combined with the appropriate byte from

$Y1$ or $Y2$, respectively.

6.3.5 SwapSoln

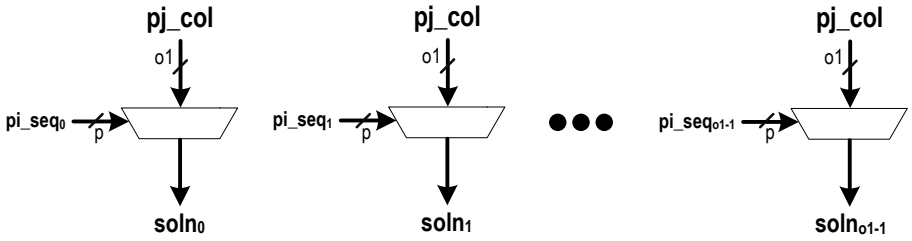

Figure 6.5: *SwapSoln* Top-Level Diagram

The top-level diagram for *SwapSoln* is shown in Fig. 6.5. It includes multiplexers controlled by the corresponding byte in *pi_seq*. The outputs $soln_i$ correspond to the ordered solution set.

Chapter 7: Binarized Systems

7.1 Matrix Binarization Theorem

Let M be a $n \times n$ matrix and z is the solution vector both with coefficients defined in $\frac{GF(2^t)}{r(x)}$ and using the standard polynomial basis, where $r(x)$ is the reduction polynomial. There exists an equivalent binary matrix, M_b, with elements in $GF(2)$ and of dimension $(n \times t) \times (n \times t)$, such that M_b has a binarized solution that is equivalent to the solution of M.

7.1.1 Corollary

This leads to an expansion factor of t in terms of additional time (clock cycles) required to solve M_b and the space requirement stays the same. However, solving the system only requires XOR/AND operations, without the need for inversion and normalization, and this can significantly reduce the critical path size. However, there is overhead required to perform the transformation in to the binarized form, which can be preprocessed or performed on the fly.

7.1.2 Proof

Binarization of an $n \times n$ matrix, M, with coefficients in $GF(2^t)$ and in the standard polynomial basis:

Let $i \in \{0 \ldots n-1\}$ and $j \in \{0 \ldots t-1\}$ and $\rho \in M$ such that $\rho = a_i b_i$, where $a_i \in GF(2^t)$ is the coefficient in M and $b_i \in GF(2^t)$ is the unknown variable. Therefore, we have $a_i = (a_{i,0} + a_{i,1}x + \cdots + a_{i,t-1}x^{t-1})$ and $b_i = (b_{i,0} + b_{i,1}x + \cdots + b_{i,t-1}x^{t-1})$, and $a_i b_i$ mod $r(x) = c_i = (c_{i,0} + c_{i,1}x + \cdots + c_{i,t-1}x^{t-1})$, where $a_{i,j}, b_{i,j} \in GF(2)$. Multiplication

followed by reduction using $r(x)$ normally produces a value $c_{i,j} \in GF(2)$, but since $b_{i,j}$ are unknown bits of b_i, only an expression dependent on $b_{i,j}$ can be derived. This expression is the basis for the binary transformation, or binarization, using r(x), and can be expressed as t equations for each $c_{i,j}$ as shown in (7.1);

$$\begin{aligned} c_{i,0} &= h_0(b_{i,0}, b_{i,1}, \ldots, b_{i,t-1}) \\ c_{i,1} &= h_1(b_{i,0}, b_{i,1}, \ldots, b_{i,t-1}) \\ &\ldots \\ c_{i,t-1} &= h_{t-1}(b_{i,0}, b_{i,1}, \ldots, b_{i,t-1}) \end{aligned} \tag{7.1}$$

where h_i is a binary function made up of binary coefficients with t variables, denoted $b_{i,j}$. Additionally, the coefficients, $c_{i,j}$, are made up of summations (XOR) containing one or more known bits from $a_{i,j}$, denoted $a_{i,j,k}$, and are dependent on r(x). For example, for any given $c_{i,j}$ we can write $c_{i,j} = a_{i,j,t-1} \times b_{i,t-1} + a_{i,j,t-2} \times b_{i,t-2} + ... + a_{i,j,1} \times b_{i,1} + a_{i,j,0} \times b_{i,0}$. Applying binarization to each polynomial in M allows for grouping like terms that amounts to adding corresponding $c_{i,j}$ terms, as shown in (7.2).

$$\begin{aligned} p_i^{(k)} &= (c_{0,0} + c_{1,0} + \cdots + c_{n-1,0}) \\ &+ (c_{0,1} + c_{1,1} + \cdots + c_{n-1,1})x + \\ &\ldots \\ &+ (c_{0,t-1} + c_{1,t-1} + \cdots + c_{n-1,t-1})x^{t-1} \end{aligned} \tag{7.2}$$

Equivalently, we can write $p_i^{(k)} = d_{i,0} + d_{i,1}x + \cdots + d_{i,t-1}x^{t-1} = z_i = z_{i,0} + z_{i,1}x + \cdots +$

$z_{i,t-1}x^{t-1}$, where z_i is the corresponding element in the solution vector. Equating each $d_{i,j}$ to the corresponding $z_{i,j}$ by using the standard polynomial form, we can derive t equations.

Applying this to all n polynomials, we can obtain M_b, containing $n \times t$ equations in $n \times t$ binary variables to equivalently represent M.

7.2 Practical Example and Testing

In this section, the result of converting a 5x5 matrix with elements in $GF(2^8)$ in to a binary matrix via the binarization method will be shown. In order to produce the following results Python scripts were used that can solve any matrix in both $GF(2^8)$ and $GF(2)$ using Algorithm 1.

Starting with the following matrix, M;

$$\begin{bmatrix} 1b & 43 & d6 & 42 & 41 & e6 \\ 13 & 4d & 5e & a9 & 17 & 81 \\ 12 & 83 & 79 & 02 & 17 & df \\ f3 & 67 & f2 & 5d & 67 & f5 \\ 1e & 98 & f6 & 39 & 97 & ef \end{bmatrix}$$

Algorithm 1 produces the matrix shown below with the solution in the last column;

$$\begin{bmatrix} 01 & 00 & 00 & 00 & 00 & 4f \\ 00 & 00 & 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 & 00 & 25 \\ 00 & 00 & 00 & 01 & 00 & 22 \\ 00 & 00 & 00 & 00 & 01 & 31 \end{bmatrix}$$

The corresponding pi_seq signal is;

$$\begin{bmatrix} 0 & 2 & 1 & 4 & 5 \end{bmatrix}$$

Reordering the solution column based on *pi_seq* produces *soln*;

$$\begin{bmatrix} 4f & 25 & 00 & 22 & 31 \end{bmatrix}$$

Matrix binarization produces M_b;

$$
\begin{bmatrix}
0&0&1&1&1&0&0&0&1&0&1&0&1&0&0&0&0&1&0&1&1&1&0&0&1&0&1&0&1&0&1&1&1&0&1&0&1&0&1\\
1&0&0&1&1&1&0&0&1&1&0&1&0&1&0&0&0&0&1&0&1&1&1&0&0&1&0&1&0&1&0&1&1&1&0&1&0&1&1\\
0&1&1&1&0&1&1&0&0&1&0&0&0&0&0&0&0&0&1&0&0&1&0&0&1&1&0&0&0&0&0&1&0&1&0&0&0&0&1\\
0&0&1&1&1&0&1&1&1&0&1&0&0&0&0&0&0&0&0&1&0&0&1&1&0&1&1&0&0&0&0&1&0&1&0&1&0&0&0\\
0&0&0&1&1&1&0&1&0&1&0&1&0&0&0&0&0&0&0&0&1&0&0&0&1&0&1&1&0&0&0&1&1&0&1&0&1&0&0\\
1&0&1&1&0&1&1&0&0&0&0&0&0&1&0&1&0&0&1&0&1&0&0&0&0&0&1&1&0&1&0&0&0&0&0&0&0&0&1\\
1&1&1&0&0&0&1&1&1&0&1&0&1&0&1&1&1&0&1&1&0&1&0&1&0&0&1&1&0&1&0&1&0&1&0&1&0&1&1\\
0&1&1&1&0&0&0&1&0&1&0&1&0&1&0&0&1&0&1&1&1&0&0&1&0&1&0&1&0&0&1&1&0&1&0&1&0&1&0\\
0&1&1&0&1&0&0&0&0&0&0&1&1&0&1&0&0&1&1&1&0&0&0&1&1&0&0&0&1&1&1&0&0&1&0&0&0&1&1\\
0&0&1&1&0&1&0&0&1&0&0&0&1&1&0&1&1&1&0&0&1&1&0&0&1&1&0&0&0&0&1&1&0&0&1&0&0&0&0\\
1&1&1&1&0&0&1&0&1&1&0&1&1&1&0&0&0&0&1&0&1&1&1&0&1&1&1&1&1&0&1&1&1&1&1&1&0&1&0\\
1&1&1&1&1&0&0&1&1&1&1&0&1&1&1&0&0&0&0&1&0&1&1&1&0&1&1&1&1&1&0&0&1&1&1&1&1&0&1\\
1&1&1&1&1&1&0&0&1&1&1&1&0&1&1&1&0&0&0&0&1&0&1&1&1&0&1&1&1&1&1&0&0&1&1&1&1&0&0\\
1&0&0&1&0&1&1&0&1&1&1&0&0&0&0&1&0&1&1&1&0&1&1&1&1&1&0&1&1&1&0&1&1&1&0&1&0&1&1&1&0\\
1&0&1&0&0&0&1&1&0&1&1&0&1&0&1&0&1&1&0&0&1&0&0&1&1&1&0&0&0&1&1&0&0&0&1&0&0&0&1&1&0\\
1&1&0&1&0&0&0&1&0&0&1&1&0&1&0&1&1&1&1&0&0&1&0&0&1&1&1&0&0&1&1&0&0&1&1&1&0&0&1&1\\
1&1&1&0&1&0&0&0&1&1&1&0&1&0&1&1&1&0&1&1&0&1&1&0&1&1&1&0&0&1&1&1&0&1&1&1&1&0&0&0\\
1&1&1&1&0&1&0&0&0&1&1&1&1&1&0&1&1&0&0&0&0&1&0&0&0&1&1&1&1&1&0&1&1&1&1&0&1&1&1&1\\
1&1&1&1&1&0&0&0&1&1&0&0&1&0&1&1&1&1&1&0&1&1&0&0&0&0&1&0&0&0&1&1&1&1&1&0&1&1&1&0&1\\
1&0&0&1&0&0&1&0&0&0&1&0&0&1&0&1&1&1&0&0&0&1&0&0&0&0&0&1&0&1&1&0&1&0&1&1&1&1&1&1\\
1&0&1&0&0&0&0&1&1&1&0&1&0&1&1&1&0&0&0&1&1&0&1&0&0&0&0&0&0&0&0&1&0&0&1&0&0&0&1&1&1\\
1&1&0&1&0&0&0&0&1&1&1&0&1&0&1&1&1&0&0&0&1&1&0&1&1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&1&1\\
1&1&1&0&0&0&1&1&1&1&0&1&1&1&0&0&1&1&0&0&0&1&1&1&0&1&1&0&0&1&0&1&1&0&1&1&1&1&0&1\\
0&1&1&1&0&0&0&1&1&1&0&1&1&1&0&0&1&1&0&0&0&1&0&1&0&1&1&0&0&1&1&1&1&0&1&1&1&1&1&1\\
1&1&0&1&1&0&1&1&1&0&1&0&1&0&0&1&1&1&1&1&0&1&1&0&0&0&1&1&1&0&1&0&1&0&1&0&0&1&1\\
1&1&1&0&1&1&0&1&0&1&0&1&0&0&1&1&1&1&1&1&0&1&0&0&0&0&1&1&1&1&0&1&0&1&0&1&0&0&1\\
0&1&1&1&0&1&1&0&0&0&1&0&1&0&1&0&0&1&1&1&1&1&0&1&0&0&0&1&1&1&0&0&1&0&1&0&0&1&0&0\\
1&1&0&1&1&0&0&0&0&1&0&0&1&0&1&1&1&0&1&1&1&0&0&1&1&1&1&0&0&0&1&0&1&0&0&1&0&1&1&1\\
1&0&0&0&1&1&1&1&0&1&1&1&0&1&1&1&0&0&0&1&1&0&1&1&1&0&0&1&0&1&0&0&1&1&1&1&0&1&1&0\\
1&1&0&0&0&1&1&1&0&1&1&1&0&1&1&1&0&0&0&1&1&0&0&1&1&0&0&1&0&1&1&0&1&1&1&1&0&1&1&1\\
0&0&0&1&1&0&0&0&1&1&0&0&1&1&0&1&1&1&0&0&0&0&1&1&1&0&1&0&1&1&0&0&1&1&1&1&1&0&1&1\\
1&0&0&0&1&1&0&0&0&1&1&0&0&1&1&0&0&0&0&1&0&1&0&1&1&0&0&1&1&1&1&1&0&1&1&1&1&0&1&\\
0&1&0&1&1&1&1&0&1&1&1&1&1&1&0&1&1&1&0&0&1&1&1&0&0&0&0&1&1&1&0&1&0&0&0&0&1&0&1\\
1&0&1&0&1&1&1&1&1&1&1&1&1&1&0&1&1&1&1&0&0&1&0&1&0&0&0&0&1&1&1&0&1&0&0&0&0&1&0\\
1&1&0&1&0&1&1&1&0&1&1&1&1&1&1&0&1&1&1&1&0&0&0&0&1&0&0&0&0&1&1&1&0&1&0&0&0&0&1\\
1&1&1&1&0&0&1&1&1&1&1&0&0&1&0&1&0&0&1&1&1&0&1&0&0&1&1&1&0&0&0&0&0&1&0&1&0&1&1\\
0&1&1&0&0&0&0&1&0&0&1&1&0&1&0&0&0&0&0&1&1&0&1&1&0&0&1&0&1&1&1&0&1&1&1&0&1&1&1\\
0&0&1&1&0&0&0&0&1&0&0&1&1&0&1&0&1&0&0&0&0&1&1&0&0&1&0&1&1&1&1&1&0&1&1&1&0&1&1&1
\end{bmatrix}
$$

Using a swapless pivot method, similar to Algorithm 1, produces the following matrix with the binarized solution in the last column;

$$\begin{bmatrix}
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{bmatrix}$$

The corresponding pi_seq_b signal is;

$$\begin{bmatrix} 1 & 2 & 0 & 4 & 5 & 6 & 3 & 7 & 17 & 16 & 19 & 20 & 21 & 22 & 23 & 18 & 10 & 8 & 9 & 12 & 11 & 14 & 15 & 13 & 24 & 26 & 27 & 25 & 30 & 31 & 28 & 29 & 32 & 33 & 35 & 36 & 37 & 34 & 39 & 38 \end{bmatrix}$$

Reordering the solution column based on pi_seq_b produces;

$[0\,1\,0\,0\,1\,1\,1\,1\,0\,0\,1\,0\,0\,1\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,1\,0\,0\,0\,1\,0\,0\,0\,1\,1\,0\,0\,0\,1]$

Grouping t bits and using hex notation we obtain $soln_b$;

$[4f\ 25\ 00\ 22\ 31]$

Therefore, it has been shown that the solution for M_b is equivalent to the solution for M (ie $soln = soln_b$).

7.3 Discussion

Once M_b is obtained, LSS must change accordingly. Let the versions of PE and PivotCalc that are able to process M_b be called PE_b and PivotCalc_b, respectively. PE_b is in an array, and contains only a single AND gate and a single XOR gate with control multiplexers as shown in 7.1. PivotCalc_b has to support only two control modes: no change and eliminate, which removes any control dependency of the row operation on the current pivot row, represented by p_i. However, the pivot row, $pivot\_row$, will still be used as the row added to all PEs in elimination mode. Additionally, this requires a signal like $neq0_i$, but without the en_i, and it shares the same comparators used in the pivot calculation.

Extending the binarization method to apply to *Affine Transformation* and *Polynomial Evaluation* is possible as well, and will require a change to the design of the original binarized processing element, PE_b. Additionally, the binary transformation could be performed by preprocessing the keys, which would allow AT, PE, and SS to be performed using only XOR/AND gates. However, this will be left for future work and discussed further in Section 9.2.

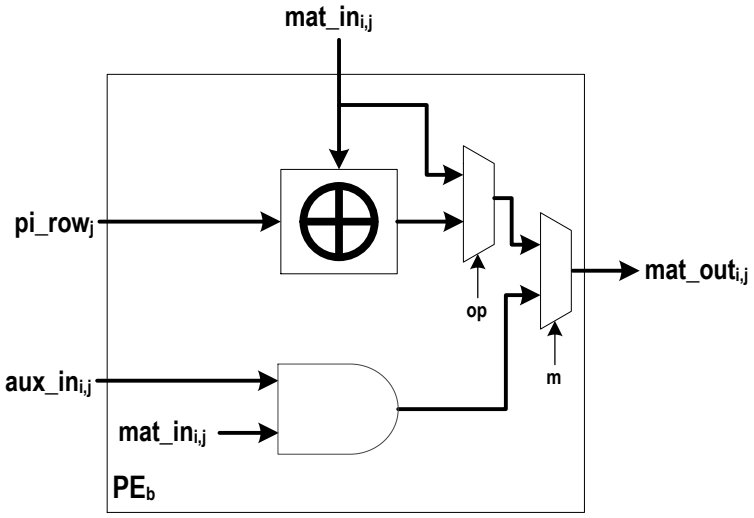

Figure 7.1: Processing Element for M_b

Chapter 8: Implementation Results

All results were generated for the Xilinx Virtex 7 FPGA (tbd).

The full system implementations for 80-bit security, UOV-80 and Rainbow-80, and 128-bit security, Rainbow-128 are shown in Table 8.1. For Rainbow-80, the signature generation time is 1450 ns, compared to 3120 ns from [17]. Not only does UOV have larger key sizes, but a larger system solver requirement, which comparatively significantly increases it's area and decreases the maximum frequency.

Table 8.1: Full Implementation Results

Algorithm Name	**N**	***LUTs***	***FFs***	***Sign Time*** (μs)	***Verify Time*** (μs)	***PubKey Load Time*** (μs)	***PrivKey Load Time*** (μs)
Rainbow-80	12	58,942	14,103	1.45	1.27	9.10	11.83
Rainbow-128	20	168,728	41,273	3.70	3.58	66.06	91.48
UOV-80	28	334,591	76,217	3.68	2.64	127.96	122.64

8.1 Component Results

Implementation results for system solvers are shown in Table 8.2. The N column corresponds to the sizes of $N \times N$ system solvers, which correspond to the different oil parameter sizes shown in Table 3.1 and Table 3.2.

Implementation results for components in the system solver are shown for Rainbow-80 and Rainbow-120 in Table 8.3 and Table 8.4, respectively.

Table 8.2: Linear System Solver results

N	***LUTs***	***FFs***	***Max Freq (MHz)***
12	37,448	78	240
20	99,116	125	240
28	205,712	125	240

Note: N > 28 was not implemented.

Table 8.3: Rainbow-80 Component Breakdown

Name	***LUTs***	***FFs***	***Max Freq (MHz)***
PivotCalc	62	76	>357
I	67	0	>357

Table 8.4: Rainbow-120 Component Breakdown

Name	***LUTs***	***FFs***	***Max Freq (MHz)***
PivotCalc	113	125	>357
I	67	0	>357

8.2 Pipelining Components

Pipelining components can increase maximum frequency of the design while using the same area and using registers. In the case of FPGAs these registers are already available in the device fabric. Unfortunately, the PE elements cannot be pipelined since the next clock cycle results for *System Solving* require the complete computation of the PE Array in order to select the pivot row. However, the $RowSum$ component may be pipelined, and this will be helpful for larger paramter sets. Additionally, the size of $RowSum$ should scale the same as the number of terms in polynomials used in UOV or Rainbow. However, pipelining was not needed in these implementations.

8.3 Binarization Results

The results for using PE_b are shown in Table 8.5. It is clear that PE_b takes much less area, however, it will require t^2 more processing elements. However, the PE Array would still require less area.

Table 8.5: Binarized PE

Name	***LUTs***	***FFs***	***WNS (ns)***
PE_b	1	0	2.28
PE	278	0	0.01

Chapter 9: Conclusions

9.1 Summary

In this thesis, we have demonstrated a high-speed architecture capable of executing the UOV and Rainbow signature schemes. We introduced a novel pivot calculation circuit, *PivotCalc*, that helped to produce a regular PE array structure. Furthermore, the PE array is controlled by a memory-based microprogramming technique to set the correct array operation and also significantly reduces multiplexing. The architecture is able to generate signatures and perform verification more than twice as fast as the previous state-of-the art high speed implementation. As shown in Table 6.1, it takes only 87 clock cycles to perform a Rainbow-80 signature generation compared to 198 clock cycles by [17]. This speed up is due to registering VV variables and because the architecture reuses all multipliers in LSS. In order to schedule appropriate inputs with respect to lines of memory corresponding to the key, a novel method was created to generate memory-based sequences that are used to perform appropriate block multiplications followed by the corresponding additions for *Affine Transformation* and *Polynomial Evaluation*.

By removing the multiplexers in the design we have greatly reduced the dependence of the critical path to the size of the system solver, LSS. Furthermore, the regular structure of the PE array allows to constrain the critical path to that of a PE. However, *PivotCalc* may grow larger due to the use of parameter dependent encoders and decoders. The size of inputs in to these components is only o_ℓ and $\lceil log_2(o_\ell) \rceil$, respectively, so we could expect a linear scaling, which is much better in contrast to the previous exponential increase in multiplexers. The reduction in critical path allows for utilization of larger systems solvers in high-speed implementations required by larger parameters sets offering more security. Additionally, the complexity of *SeqGen* and *RowSum* may be reduced, and it should be noted that MB

eliminates the need for *RowSum*. A design space exploration will be conducted to assess the best trade off for throughput/area adjusting the number of rows in the PE array, and consequently, the number of multipliers in the system. The use of composite field multipliers may further reduce the critical path, and pipelining may also help.

We introduced a binarization method that is applied to matrices with elements in $GF(2^t)$, and show that it has a potential to reduce the critical path of system solvers required for Multivariate systems. Traditionally, all system solvers used in Multivariate systems are explicitly designed or extended to operate on elements in $GF(2^t)$, and this requires additional area overhead of the inverse component and multipliers. For the first time, we eliminate this requirement for all Multivariate schemes, and demonstrate a circuit capable of operation using only AND/XOR gates, which will support either SS only or AT, SS and PE, respectively. This allows for a large number of system solver architectures, aimed at $GF(2)$ matrices, to be useful in Multivariate schemes.

9.2 Future Work

Future work will include further reduction in area for high-speed implementations by reducing the complexity of *SeqGen* and *RowSum* and using a limited number of rows (ie $< o_\ell$) in *LSS*. This will result in a reduction of multipliers available at the cost of an increase in the latency required for signature generation and verification, effectively trading off area for latency. Additionally, the critical path can be improved by exploring composite field multipliers, such as in [26], and implementing the full design with pipelined components.

Additionally, MB eliminates *RowSum*, but still requires *SeqGen*. Implementing PE_{b1} would require additional multiplexers for the dual mode of operation (ie system solving and block multiplication), and redesigning the PE array to support adder reuse in order to add like terms without *RowSum*. The pattern of addition among PEs can be controlled by multiplexers, and it can be generated by *SeqGen*. This new PE array also can drastically speed up the process of *Affine Transformation* and *Polynomial Evaluation*, which are still

more than 70% of the execution time, at the cost of area. The speedup in AT and PE would be possible at the cost of increased area due to a larger PE array. However, this would offset the increase in clock cycles for SS, which would be t times larger than LSS, and decrease the latency and the critical path.

Bibliography

Bibliography

[1] National Security Agency. (2015, Aug) Cryptography today. [Online]. Available: http://www.tinyurl.com/SuiteB

[2] D. J. Bernstein, "Introduction to Post-Quantum Cryptography," in *Post-quantum cryptography*. Springer, 2009, pp. 1–14.

[3] National Institute of Standards and Technology. (2016, Apr) Report on post-quantum cryptography. [Online]. Available: https://csrc.nist.gov/publications/detail/nistir/8105/final

[4] ——. (2016, Dec) Post-quantum crypto project. [Online]. Available: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography

[5] R. Pellikaan, "Evaluation of public-key cryptosystems based on algebraic geometry codes," 01 2011.

[6] J. Patarin, "The Oil and Vinegar Signature Scheme." presented at the Dagstuhl Workshop on Cryptography.

[7] A. Kipnis and A. Shamir, "Cryptanalysis of the Oil and Vinegar Signature Scheme," in *Annual International Cryptology Conference*. Springer, 1998, pp. 257–266.

[8] J. Ding and D. Schmidt, "Rainbow, a New Multivariable Polynomial Signature Scheme," in *International Conference on Applied Cryptography and Network Security*. Springer, 2005, pp. 164–175.

[9] J. Ding, J. Gower, and S. Dieter, "Multivariate Public Key Cryptosystems." Springer.

[10] P.-A. Fouque, L. Granboulan, and J. Stern, "Differential Cryptanalysis for Multivariate Schemes," in *Eurocrypt*, vol. 3494. Springer, pp. 341–353.

[11] J. Ding, B.-Y. Yang, C.-H. Chen, M.-S. Chen, and C.-M. Cheng, "New Differential-Algebraic Attacks and Reparametrization of Rainbow," in *Applied Cryptography and Network Security*. Springer, 2008, pp. 242–257.

[12] M.-S. Chen, W.-D. Li, B.-Y. Peng, B.-Y. Yang, and C.-M. Cheng, "Implementing 128-bit Secure MPKC Signatures," in *Cryptology ePrint Archive, Report 2017/636, 2017. http://eprint.iacr.org/2017/636, 2017.*

[13] A. Petzoldt, "Selecting and reducing key sizes for multivariate cryptography," Ph.D. dissertation, Dissertation, Darmstadt, Technische Universität Darmstadt, 2013.

[14] D. Moody, R. Perlner, and D. Smith-Tone, "An Asymptotically Optimal Structural Attack on the ABC Multivariate Encryption Scheme," in *International Workshop on Post-Quantum Cryptography*. Springer, 2014, pp. 180–196.

[15] Okayama University of Science. (2015, Jun) Fukuoka MQ Challenge. [Online]. Available: https://www.mqchallenge.org/

[16] S. Balasubramanian, H. W. Carter, A. Bogdanov, A. Rupp, and J. Ding, "Fast Multivariate Signature Generation in Hardware: The Case of Rainbow," in *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*. IEEE, 2008, pp. 25–30.

[17] S. Tang, H. Yi, J. Ding, H. Chen, and G. Chen, "High-speed Hardware Implementation of Rainbow Signature on FPGAs," in *International Workshop on Post-Quantum Cryptography*. Springer, 2011, pp. 228–243.

[18] A. Bogdanov, T. Eisenbarth, A. Rupp, and C. Wolf, "Time-Area Optimized Public-Key Engines: MQ-Cryptosystems as Replacement for Elliptic Curves?" in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2008, pp. 45–61.

[19] B. Hochet, P. Quinton, and Y. Robert, "Systolic Gaussian Elimination over GF(p) with Partial Pivoting," *IEEE Transactions on Computers*, vol. 38, no. 9, pp. 1321–1324, 1989.

[20] C.-L. Wang and J.-L. Lin, "A Systolic Architecture for Computing Inverses and Divisions in Finite Fields GF(2^m)," *IEEE Transactions on Computers*, vol. 42, no. 9, pp. 1141–1146, 1993.

[21] A. Bogdanov, M. C. Mertens, C. Paar, J. Pelzl, and A. Rupp, "SMITH - A Parallel Hardware Architecture for Fast Gaussian Elimination over GF(2)," in *Workshop on Special-purpose Hardware for Attacking Cryptographic Systems (SHARCS 2006)*.

[22] A. Rupp, T. Eisenbarth, A. Bogdanov, and O. Grieb, "Hardware SLE Solvers: Efficient Building Blocks for Cryptographic and Cryptanalytic Applications," *Integration, the VLSI journal*, vol. 44, no. 4, pp. 290–304, 2011.

[23] A. Ferozpuri, F. Farahmand, M. U. Sharif, J.-P. Kaps, and K. Gaj. (2016, Oct) Hardware API for Post-Quantum Public Key Cryptosystems. [Online]. Available: https://cryptography.gmu.edu/athena/PQC/PQC_HW_API.pdf

[24] A. Kipnis, J. Patarin, and L. Goubin, "Unbalanced Oil and Vinegar Signature Schemes," in *EUROCRYPT'99*.

[25] T. Zhang and K. K. Parhi, "Systematic Design of Original and Modified Mastrovito Multipliers for General Irreducible polynomials," *IEEE Transactions on Computers*, vol. 50, no. 7, pp. 734–749, 2001.

[26] H. Yi and W. Li, "Fast Three-Input Multipliers over Small Composite Fields for Multivariate Public Key Cryptography," *International Journal of Security and Its Applications*, vol. 9, no. 9, pp. 165–178, 2015.

Curriculum Vitae

Ahmed Q. Ferozpuri graduated from Herndon Highschool, Herndon, Virginia, in 1999. He graduated Cum Laude from George Mason University in 2007 with a B.S. in Electrical Engineering and a B.S. in Computer Science. He was employed by various DoD contractors in the Northern Virginia area as an Electronics, Computer, Software and Radar engineer. He has experience with hardware and software design - developing products in websites, computational software, Graphical User Interfaces (GUI), embedded systems and hardware with projects related to networking, cryptography, signal processing and machine learning.