

POLITECNICO DI TORINO

Master of Science in Electronic Engineering

Master Thesis

# Hardware acceleration for post-quantum cryptography



**Supervisors**

Prof. Guido Masera

Prof. Maurizio Martina

**Candidate**

Flavio Tanese

December 2018



## Abstract

Communication security of today heavily relies on the assumption that some mathematical problems are extremely difficult to solve and thus breaking encryptions based on such problems requires a very long time. While such encryptions are secure now, the probable diffusion of quantum computers in the foreseeable future makes the initial assumption fall short: quantum computations are efficient at breaking the most widespread algorithms in use. Post-quantum cryptographic systems are based on problems that are not (or marginally) affected by the peculiarity of quantum computing: AES[3] and many other hashing functions fall in this category, with quantum operations just moving the problem from  $O(N)$  to  $O(\sqrt{N})$ , with  $N$  being the number of operations needed to find a solution. This is effectively countered by using double the number of bits and squaring the complexity. Other proposals are based on variations of error-correcting codes used in data transmission, so that the data is encoded and errors are purposely introduced in the encrypted version. With no a priori knowledge on the location of such errors, reverse-engineering the generation matrix becomes a very arduous task, making the system de facto equivalent to the prime-based asymmetric key system in use today but without the vulnerability to quantum attacks. This work is focused on a hardware implementation of such a system, for use in low power applications that are likely to generate the bulk of encrypted traffic in the near future.

# Contents

<b>1</b>	<b>Basics of cryptography</b>	<b>3</b>
1.1	Symmetric and asymmetric ciphers . . . . .	4
1.2	RSA . . . . .	6
1.3	Shor's algorithm . . . . .	7
<b>2</b>	<b>The LEDAcrypt cryptosystem</b>	<b>8</b>
2.1	QC-LDPC codes . . . . .	8
2.2	LEDAcrypt's keys . . . . .	9
2.3	Encryption and decryption . . . . .	10
<b>3</b>	<b>Hardware implementation of LEDAcrypt decryption</b>	<b>12</b>
3.1	Assumptions on memory . . . . .	13
3.2	System parameters . . . . .	13
<b>4</b>	<b>Key reconstruction</b>	<b>15</b>
4.1	Circulant block multiplication . . . . .	15
4.1.1	Out-of-order result and modulo $p$ implementation . . .	17
4.1.2	Result sorting . . . . .	19
4.1.3	Modulo 2 on compressed matrices . . . . .	20
4.2	Circulant block sum . . . . .	21
4.2.1	Memory movement . . . . .	22
4.3	Quasi-cyclic multiplication . . . . .	22
<b>5</b>	<b>Vector by matrix multiplication (and vice-versa)</b>	<b>25</b>
5.1	Vector by circulant matrix . . . . .	25
5.2	Circulant matrix by vector . . . . .	27
5.3	$x$ by $\mathbf{L}^T$ . . . . .	28
5.4	$\mathbf{H}^T$ by $s^T$ . . . . .	29
5.5	$\mathbf{Q}^T$ by $\Sigma^T$ . . . . .	30
5.6	$e$ by $\mathbf{H}^T$ . . . . .	30

<b>6</b>	<b>Error update</b>	<b>32</b>
6.1	Peak search . . . . .	32
6.2	Row extraction from compressed matrix . . . . .	33
6.3	Vector plus compressed row . . . . .	33
<b>7</b>	<b>Main loop state machine</b>	<b>35</b>
7.1	Design modularity and shared resources . . . . .	36
<b>8</b>	<b>Conclusions</b>	<b>38</b>
<b>A</b>	<b>Source code</b>	<b>40</b>
A.1	Key reconstruction . . . . .	40
A.1.1	Sorting . . . . .	43
A.1.2	Circulant sum . . . . .	47
A.1.3	Memory copy . . . . .	50
A.2	Vector by matrix . . . . .	52
A.2.1	Vector by circulant . . . . .	52
A.2.2	$x$ by $\mathbf{L}^T$ . . . . .	55
A.2.3	$\mathbf{H}^T$ by $s^T$ . . . . .	56
A.2.4	$\mathbf{Q}^T$ by $\Sigma^T$ . . . . .	58
A.2.5	$e$ by $\mathbf{H}^T$ . . . . .	60
A.3	Error update . . . . .	62
A.4	Loop control and message computation . . . . .	66
A.5	Top module . . . . .	71

# Chapter 1

## Basics of cryptography

Cryptography comes from two ancient Greek words that more or less translate to “hidden writing”, originally with the objective of having a reliable way to deliver military orders through messengers without the enemy understanding intercepted ones[4]. While this specific application proved by far the biggest drive to cryptography up to recent times, this “hidden writing” capability is now heavily used by civilians too due to the vast amount of sensitive information that is transmitted through potentially unsecure channels.

Traditionally, the agents in cryptography examples are called Alice (A, the sender), Bob (B, the recipient) and Eve (E, the eavesdropper). Alice and Bob use cryptography so they can communicate without Eve being able to understand the message, even though Eve might intercept the code (from here on, “code” is used to refer to the encrypted version of the message). Since it is assumed that Eve knows the code, Bob must have some information not contained in the code that can be used to get the message back from it: this information is known as the “key” (figure 1.1).

In the oldest and simplest ciphers, the number of possible keys was usually quite small, so that Eve could simply try them all and see which key yielded a message that made sense. This was only effective as long as Eve did not have a clue about the mechanism of the cipher, so that Alice and Bob mostly relied on what is called “security by obscurity” and had to keep the cipher itself secret.

Since a secret mechanism does not scale up well with many possible recipients and devising a new cipher for each recipient would be a daunting task (that still requires a secure channel anyway), modern ciphers are public, while relying on other features to protect messages. These features arise from particular mathematical properties and are meant to prevent anyone not having the key from decrypting the code, while the possible number of keys is so big that brute forcing (i.e. trying them all one by one) is pointless.

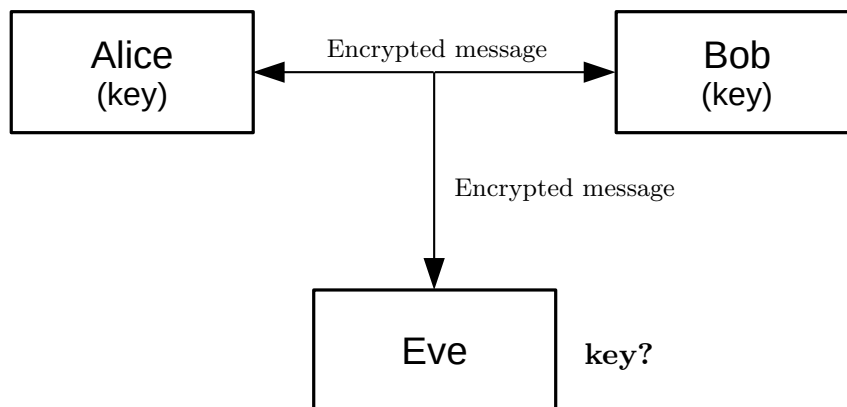


Figure 1.1: Alice, Bob and Eve

## 1.1 Symmetric and asymmetric ciphers

Ciphers in use today fall into two broad categories: symmetric and asymmetric ciphers. The former category is made up with all systems that require Alice and Bob to know the same key (hence “symmetric”), and Eve not to know the key: it requires a secure channel for sharing the key between Alice and Bob in the first place (figure 1.2). The latter is made up with systems that have Bob know a private key nobody else knows (hence “asymmetric”), and everyone know Bob’s public key that is used to encrypt the message (figure 1.3). The system is then conceived in such a way that only Bob’s private key can decrypt what was encrypted with the public key. Asymmetric ciphers, while not extremely complicated in their most basic form, are much more recent than symmetric ciphers: the earliest military implementation was devised in 1973, while the first civilian algorithm dates 1976.

Symmetric ciphers are usually very simple and very fast to implement both in software and hardware. Unfortunately, they can only be used when Alice and Bob have a way to agree on a key without Eve intercepting it. Asymmetric ciphers let Alice and Bob communicate without any need to share their private keys, but are usually very slow and possibly quite complex. Neither of the categories can respond to the need for a massive amount of data to be transferred securely and quickly, but an asymmetric cipher can be used to send a symmetric key without Eve knowing it, and that key can then be used to encrypt and decrypt the data (figure 1.4).

The Internet itself relies on such a method for its TLS (Transport Layer Security) protocol. While TLS does not mandate any particular algorithm, the most common choice is RSA (Rivest-Shamir-Adleman, its inventors) as

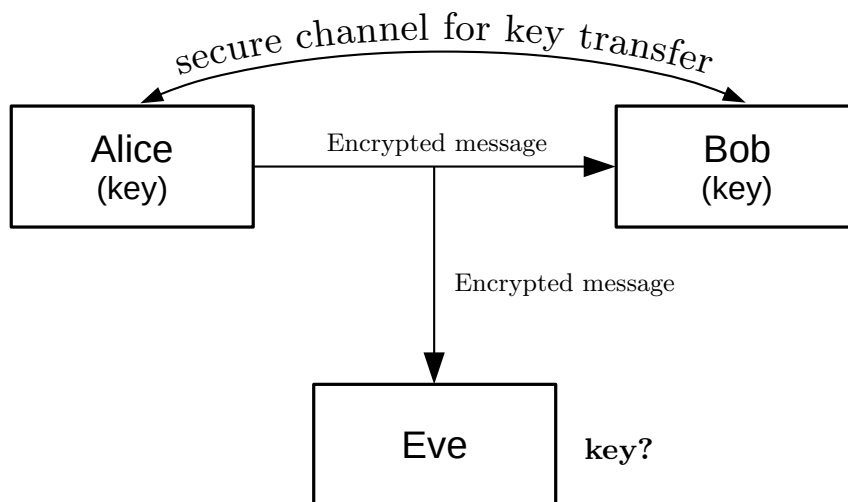


Figure 1.2: Secure channel for key transmission

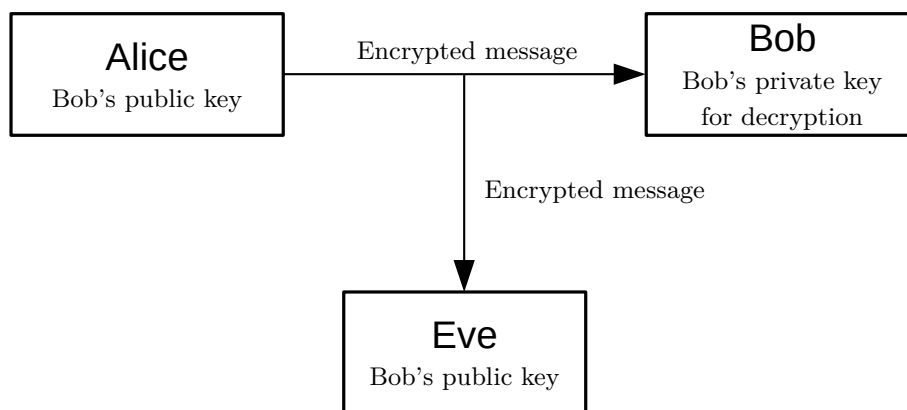


Figure 1.3: Asymmetric key not requiring a secure channel



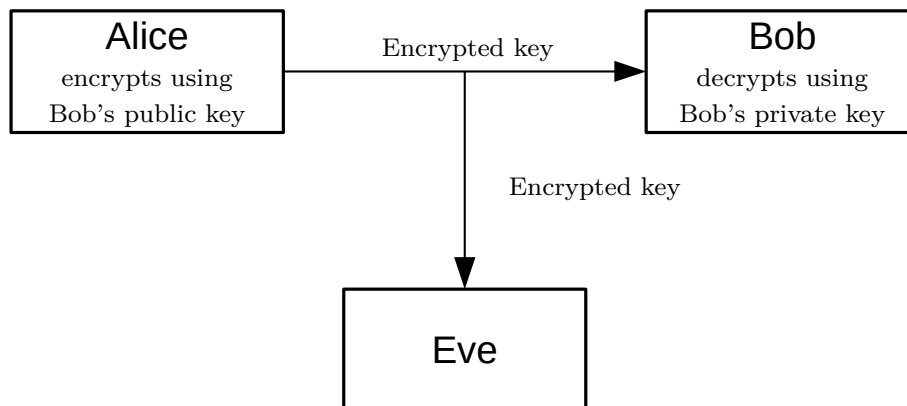


Figure 1.4: Asymmetric cypher creating a secure channel for a symmetric key

asymmetric cipher and AES (Advanced Encryption Standard) as symmetric cipher.

## 1.2 RSA

RSA[5] is a relatively simple asymmetric cypher published in 1978, just two years after the first non-classified paper on such cryptosystems by Diffie and Hellman. The simplest explanation of RSA, while not really exact, is straightforward: choose two very big prime numbers, keep them secret and provide their product as public key. Factoring such a huge number into its two prime factors is extremely demanding in terms of computational power, and whenever processors become faster and more powerful it is a simple matter of choosing even bigger starting numbers, making the algorithm extremely scalable.

The actual RSA key generation algorithm is as follows:

- Randomly pick two distinct prime numbers,  $p$  and  $q$
- Compute  $n$  as  $n = pq$
- Compute  $\lambda(n)$  as the least common multiple of  $p - 1$  and  $q - 1$
- Randomly pick an integer  $e$  that is smaller than  $\lambda(n)$  and coprime with it (no common divisors other than 1)
- Compute  $d$  such that  $de \pmod{\lambda(n)} = 1$

- Share  $n$  and  $e$  as public key, while keeping  $d$  (and technically  $n$ ) as private key

Encryption of a message  $m$  is straightforward, if computationally intensive, as  $x = m^e \pmod{n}$ .

Decryption is very similar in that  $m = x^d \pmod{n}$ .

### 1.3 Shor's algorithm

From the description of the RSA algorithm it can be noted that once an attacker is able to get  $p$  and  $q$  he can also easily compute  $d$  using the same procedure that is used for key generation. While it is not proved that computing  $d$  requires explicitly factoring  $n$ , no known method that exploits the availability of  $e$  has been published. It is thus paramount that getting  $p$  and  $q$  from  $n$  be extremely time consuming: classical algorithms for factorization require exponential time, and, while the shorter 1024-bit RSA keys might be breakable given enough time and resources, the longer keys up to 4096 bits are still impregnable to any foreseeable attack.

In 1994 Peter Shor, at the time working at Bell Laboratories and now professor of mathematics at MIT, devised an algorithm[6] that can efficiently factor any number that is not an integer power of a prime number. Since a requirement of RSA is that  $p$  and  $q$  are prime and different, the condition for applying Shor's algorithm holds. While the inner workings of the algorithm are out of the scope of this thesis, the general idea of the algorithm is that through quantum operations it is possible to obtain the period of the function  $f(x) = a^x \pmod{n}$ , which is in turn directly related to  $p$  and  $n$ . The algorithm requires  $2\log_2(n)$  quantum bits to be effective, and while this amounts to several thousands qubits (a far stretch from the 50 qubits available to the most powerful devices at the beginning of 2018) the number is not inherently prohibitive assuming quantum computing will undergo a similar evolution as classical computing[7].

As a direct consequence of this perceived danger, researches have been devising alternative cyphering systems that are supposedly robust to attacks coming from future quantum computers.

## Chapter 2

# The LEDAcrypt cryptosystem

The LEDAcrypt cryptosystem, developed by Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi and Paolo Santini, is actually not one cryptosystem but two. The first one, LEDAkem[1], is a Key Encapsulation Mechanism, while the second one, LEDApkc, is a Private Key Cryptosystem. They are however very similar in concept and implementation, so they will be treated together from here on.

LEDAcrypt is built on the McEliece cryptosystem[8], that uses linear codes. The basic idea behind this cryptosystem is that decoding a generic error-correcting code without knowing the decoding function is NP-hard. This in turn requires being able to give a public key for anyone to encrypt a message, while the private key that decodes the message is kept secret and cannot be obtained from the public one. While the McEliece cryptosystem is quite robust, with no known attacks that cannot be neutralized by slight modification of the original system, it has almost never been used due to the sheer dimension of the keys it requires. A standard set of keys for a McEliece cryptosystem can be as big as 500 kb, which is an obvious setback if compared to RSA's 4 kb.

Honouring the convention used by the authors of the cryptosystem in the original paper, in this thesis vectors are row vectors unless otherwise specified and transposed vectors are column vectors.

## 2.1 QC-LDPC codes

LEDAcrypt uses QC-LDPC (Quasi-Cyclic Low-Density Parity-Check) codes, that are based on quasi-cyclic binary matrices (hence the name). Quasi-cyclic matrices are matrices having circulant blocks: each block can be completely described by its first row. With a block size  $p \times p$  and a reasonable  $p$  value,

this leads to keys more than 25,000 times smaller than they would be if they were not circulant.

These quasi-cyclic blocks, however, are also extremely sparse and binary. This property means it is possible to write, for each circulant block, the position of set elements on the first row (knowing their value is one), while everything else is assumed to be zero. A typical block is thus described with a small number of integers and takes up only a few bytes.

## 2.2 LEDAcrypt's keys

The particular code used by LEDAcrypt is made up with two matrices forming the private key, from here on called  $\mathbf{H}$  and  $\mathbf{Q}$ :

$$\mathbf{H} = [\mathbf{H}_0 \mid \mathbf{H}_1 \mid \cdots \mid \mathbf{H}_{n_0-1}] \quad (2.1)$$

$$\mathbf{Q} = \begin{bmatrix} \mathbf{Q}_{0,0} & \mid & \mathbf{Q}_{0,1} & \mid & \cdots & \mid & \mathbf{Q}_{0,n_0-1} \\ \mathbf{Q}_{1,0} & \mid & \mathbf{Q}_{1,1} & \mid & \cdots & \mid & \mathbf{Q}_{1,n_0-1} \\ \vdots & \mid & \vdots & \mid & \ddots & \mid & \vdots \\ \mathbf{Q}_{n_0-1,0} & \mid & \mathbf{Q}_{n_0-1,1} & \mid & \cdots & \mid & \mathbf{Q}_{n_0-1,n_0-1} \end{bmatrix} \quad (2.2)$$

Each block  $\mathbf{H}_i$  in (2.1) and  $\mathbf{Q}_{ij}$  in (2.2) has size  $p \times p$ , with  $p$  prime: this makes the system immune to a particular type of attack and ensures invertibility of a matrix that will need inversion to compute the public key. Parameter  $n_0$  is a small integer, that can be as small as 2. All blocks  $\mathbf{H}_i$  of  $\mathbf{H}$  have weight (number of set elements)  $d_v$ , with a standard choice being 17, while blocks of  $\mathbf{Q}$  have a weight according to the following map (which is, by the way, circulant as well):

$$\mathbf{W} = \begin{bmatrix} m_0 & \mid & m_1 & \mid & \cdots & \mid & m_{n_0-1} \\ m_{n_0-1} & \mid & m_0 & \mid & \cdots & \mid & m_{n_0-2} \\ \vdots & \mid & \vdots & \mid & \ddots & \mid & \vdots \\ m_1 & \mid & m_2 & \mid & \cdots & \mid & m_0 \end{bmatrix}$$

where  $m_i$  are again small integer values.

From matrices  $\mathbf{H}$  and  $\mathbf{Q}$  a new matrix  $\mathbf{L}$  is obtained as:

$$\mathbf{L} = \mathbf{H}\mathbf{Q} = [\mathbf{L}_0 \mid \mathbf{L}_1 \mid \cdots \mid \mathbf{L}_{n_0-1}]$$

Given a proper choice of parameters  $d_v$  and  $\underline{m} = [m_0, m_1, \dots, m_{n_0-1}]$  the inventors of the cryptosystem have proven that  $\mathbf{L}_{n_0-1}$  is invertible. This

means any possible secret key satisfying the constraints on the parameters can be used to compute a corresponding public key  $\mathbf{M}$  such that:

$$\mathbf{M} = \mathbf{L}_{n_0-1}^{-1} \mathbf{L} = [\mathbf{M}_0 \mid \mathbf{M}_1 \mid \cdots \mid \mathbf{M}_{n_0-2} \mid \mathbf{I}_p] = [\mathbf{M}_1 \mid \mathbf{I}_p]$$

The generator matrix is then obtained as:

$$\mathbf{G}' = [\mathbf{I}_{p(n_0-1)} \mid \mathbf{M}_1^T]$$

with  $\mathbf{M}_1^T$  being the transpose of  $\mathbf{M}_1$ . An important thing to notice is that  $\mathbf{M}$ , albeit dense and thus not possible to compress as much as  $\mathbf{H}$  and  $\mathbf{Q}$ , is quasi-cyclic as well. This leads to a public key of size  $p(n_0 - 1)$  bits, as the last  $p$  bits of  $\mathbf{M}$  are known by construction and  $\mathbf{G}'$  is obtained easily from  $\mathbf{M}_1$ .

## 2.3 Encryption and decryption

The ciphertext  $\underline{x}$  of size  $1 \times pn_0$  is obtained by multiplying a message  $\underline{u}$  of size  $1 \times p(n_0 - 1)$  by the generator matrix  $\mathbf{G}'$  as follows:

$$\underline{x} = \underline{u}\mathbf{G}' + \underline{e}$$

with  $\underline{e}$  being a purposely introduced error having weight  $t$  which is low enough for the code to correct with a very high chance. This is necessary because the first  $p(n_0 - 1)$  bits of  $\underline{u}\mathbf{G}'$  correspond to  $\underline{u}$  itself.

The decryption algorithm used by LEDAcrypt is a custom bit-flipping algorithm that succeeds when the syndrome of the code is null (since the fundamental property of the syndrome in linear codes is that it is only null for valid codewords, this effectively amounts to having removed the error). The starting syndrome  $\underline{s}$  is computed as

$$\underline{s}^T = (\mathbf{H}\mathbf{Q})\underline{x}^T$$

and updated with an iterating algorithm, while the error  $\underline{e}$  is initialized to a zero vector.

The main loop of decryption involves computing a vector  $\underline{R}$  such that

$$\underline{\Sigma}^{(l)} = \underline{s}^{(l-1)}\mathbf{H}$$

$$\underline{R}^{(l)} = \underline{\Sigma}^{(l)}\mathbf{Q}$$

with  $\underline{\Sigma}$  and  $\underline{R}$  being vectors of natural numbers, in contrast with every other vector and matrix which are binary.

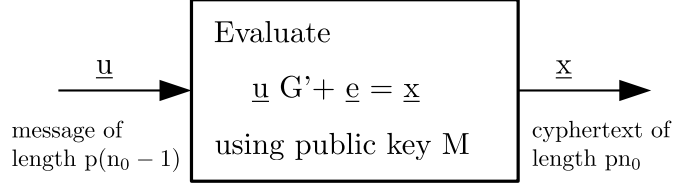


Figure 2.1: Encryption in LEDAcrypt

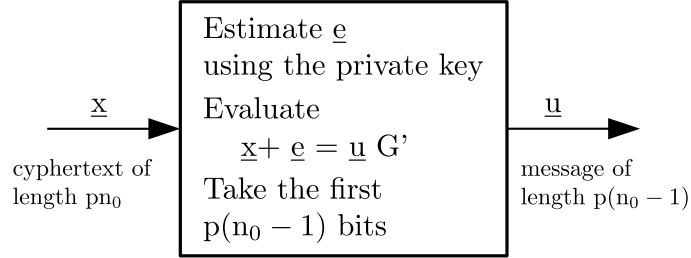


Figure 2.2: Decryption in LEDAcrypt

It is now necessary to find the positions in which  $\underline{R}^{(l)}$  is maximum, here denoted as set  $\mathfrak{J}^{(l)}$ . These positions are the ones that most likely correspond to wrong bits. Flipping bits is not done directly on the received code  $\underline{x}$ , but rather the knowledge of  $\mathbf{H}$  and  $\mathbf{Q}$  allows for direct incremental updating of  $\underline{e}$  and  $\underline{s}$ . The new value of  $\underline{e}$  is obtained as

$$\underline{e}^{(l)} = \underline{e}^{(l-1)} + \sum_{v \in \mathfrak{J}^{(l)}} \underline{q}_v$$

with  $\underline{q}_v$  being the  $v^{\text{th}}$  row of  $\mathbf{Q}$  and  $v$  being one of the indices corresponding to maximum  $\underline{R}^{(l)}$ . Having now the updated error, the updated syndrome is found as

$$\underline{s}^{(l)} = \underline{s}^{(l-1)} + \underline{e}^{(l)} \mathbf{H}^T$$

and the algorithm either terminates (due to a null syndrome or exceeding the number of permitted iterations) or starts a new cycle of the main loop.

If the algorithm terminated due to null syndrome, the error is then known as the last  $\underline{e}^{(l)}$  and it is then easy to get the message as the first part of the corrected code:

$$\begin{aligned} \underline{u} \mathbf{G}' &= \underline{x} + \underline{e}^{(l)} \\ \underline{u} &= (\underline{x} + \underline{e}^{(l)})[0 : p(n0 - 1) - 1] \end{aligned}$$

## Chapter 3

# Hardware implementation of LEDAcrypt decryption

This thesis is aimed at obtaining a hardware implementation of the decryption system described in chapter 2. The chosen Hardware Description Language was VHDL (VHSIC Hardware Description Language, with VHSIC standing for Very High Speed Integrated Circuits), as the one that the author was most familiar with at the beginning of the work, although it must be noted that a more modern alternative exists in the form of SystemVerilog. The syntaxes of these two languages, while mostly presenting clear parallelisms, are quite different and have different tradeoffs: VHDL is a very mature language, quite limited in core features, very well supported by EDA (Electronic Design Automation) tools but quite pedantic, especially in terms of typing checks and process triggers; SystemVerilog is much more lenient but only a subset of the extensive standard is supported by EDA tools and the additional flexibility necessarily implies additional risk of inadvertently making a mistake that is interpreted as a legal construct.

The reason for supporting the decryption specifically is that the operations involved in encrypting are quite cheap to perform on a general purpose processor, as it is a single pass operation consisting of copying a message, padding it with the result of a single vector-by-matrix multiplication and adding a few errors. Decrypting, on the other hand, features multiple vector-by-matrix multiplications, peak detection and vector sums in a loop which could keep the processor busy for longer than it is acceptable.

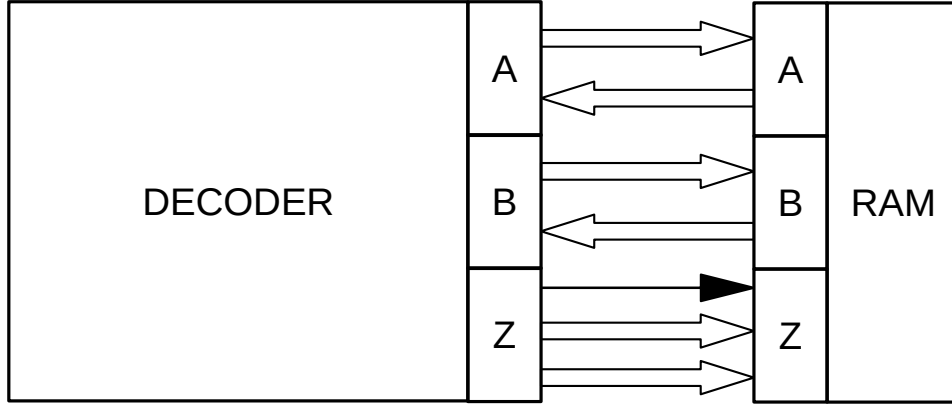


Figure 3.1: Decoder and memory

### 3.1 Assumptions on memory

An important detail is that the decryption operation is heavily limited by access to memory. This is due to the extreme reliance of the algorithm on linear algebra using very big matrices and vectors, that have to reside in some kind of RAM: while storing everything in flip-flops is theoretically possible, the parameters giving the least secure implementation would still result in 225,000 flip-flops as a conservative estimate. Because of the huge area it would require, parallel access to the entire vector is impossible: to avoid excessive restraint on the algorithm I assumed a memory that can read two values per cycle and write one is available (as in figure 3.1). This would of course be custom on-chip memory, and in case of an FPGA implementation an emulation could be achieved with parallel writing on multiple memory chips at the cost of increased storage occupation. A single read or write operation per cycle would take almost three times to complete decryption, making dedicated hardware somewhat redundant, and sharing memory with the main processor would likely defeat the purpose entirely by occupying the bus.

### 3.2 System parameters

The implementation is completely controlled by parameters, meaning that any legal combination of block size, weights and number of blocks can be implemented by plugging in the desired values. Memory mapping is automatic and has no impact, but it can be easily tweaked too to fit into a larger module featuring secure communication with the processor if need be.



Default parameters are as follows:

$n_0$	$p$	$d_v$	$\underline{m}$
2	27779	17	[4, 3]

with  $n_0$  being the number of circulant blocks in matrix  $\mathbf{H}$ ,  $p \times p$  being the size of the circulant blocks,  $d_v$  being the weight (number of set elements) of each circulant block of  $\mathbf{H}$  and  $\underline{m}$  being the first row of matrix  $\mathbf{W}$ , that is circulant and contains the weights of the blocks of  $\mathbf{Q}$ .

The implementation assumes that at the start of decryption matrices  $\mathbf{H}$  and  $\mathbf{Q}$ , making up the secret key, are loaded into memory, and that the code to decrypt,  $\underline{x}$ , is in memory as well.  $\mathbf{H}$  is stored as follows:

H BASE ADDRESS +	0	1	$\cdots$	$d_v - 1$	$d_v$	$\cdots$	$n_0 d_v - 1$
content	$H_0^T$	$H_1^T$	$\cdots$	$H_{d_v-1}^T$	$H_{d_v}^T$	$\cdots$	$H_{n_0 d_v-1}^T$

so that the set positions of the transpose of  $\mathbf{H}$  are what is actually in memory.

Similarly,  $\mathbf{Q}$  is stored as:

Q BASE ADDRESS +	0	1	$\cdots$	$m_0 - 1$	$m_0$	$\cdots$	k-1
content	$Q_{0,0}^T$	$Q_{0,1}^T$	$\cdots$	$Q_{0,m_0-1}^T$	$Q_{0,m_0}^T$	$\cdots$	$Q_{0,k-1}^T$
Q BASE ADDRESS +	$k$	$k+1$	$\cdots$	$k + m_{n_0-1} - 1$	$k + m_{n_0-1}$	$\cdots$	$2k - 1$
content	$Q_{1,0}^T$	$Q_{1,1}^T$	$\cdots$	$Q_{1,m_0-1}^T$	$Q_{1,m_0}^T$	$\cdots$	$Q_{1,k-1}^T$

$$k = \sum_{i=0}^{n_0-1} m_i$$

with  $\underline{m}$  here indicating the first row of  $\mathbf{W}^T$  for brevity, as  $\mathbf{Q}$  is also transposed.

With the default parameters, this amounts to 48 16-bit words of storage (although 15 bits would suffice, if deviating from the standard of using powers of 2 is allowed).

For ease of design it was assumed that the bits of the code  $\underline{x}$  are accessible one by one by their index, although the design does not enforce that each bit is stored in a 1-bit memory location if a custom memory is not available. This does result in a substantial waste of space, though, and the assumption is as always that we have a custom memory in our chip: in this case this would allow us to have no waste while having access to single bits.

# Chapter 4

## Key reconstruction

The first step needed to decrypt the code is obtaining the syndrome. Since

$$\underline{s}^T = \mathbf{L}\underline{x}^T \mapsto \underline{s} = \underline{x}\mathbf{L}^T$$

holds, the objective of this submodule is computing  $\mathbf{L}^T$  as

$$\mathbf{L} = \mathbf{H}\mathbf{Q} \mapsto \mathbf{L}^T = \mathbf{Q}^T\mathbf{H}^T$$

Handling traditional matrix multiplication, with  $\mathbf{H}$  having size  $27779 \times 55558$  and  $\mathbf{Q}$  having size  $55558 \times 55558$  at best, is out of question: such a calculation requires  $8.5 \cdot 10^{13}$  multiplications and about as many sums and would take ages. The particular format of  $\mathbf{H}$  and  $\mathbf{Q}$ , however, allows for a very efficient implementation.

### 4.1 Circulant block multiplication

A binary circulant matrix having only its first element set is the identity matrix, and multiplying any matrix by it results in the starting matrix. A binary circulant matrix having only its second element set circularly shifts all rows of the other operand right by one position, and so on. It follows that the multiplication between two binary circulant matrices, with one having a single set element in the first row, is another binary circulant matrix of the same size. We can then easily extend the result by expressing any binary circulant matrix as the sum of many having a single set element, and state that the product of any two binary circulant matrices is circulant: to ensure it is binary it is sufficient to perform all sums of partial products modulo 2.

It is then possible to obtain the product of two circulant blocks  $\mathbf{A}$  and  $\mathbf{B}$  as follows:

$$\mathbf{A} = \begin{bmatrix} a_0 & a_1 & \cdots & a_{m-1} \end{bmatrix}$$

$$\mathbf{B} = [b_0 \quad b_1 \quad \cdots \quad b_{n-1}]$$

$$\mathbf{AB} = [a_0 + b_0 \quad a_0 + b_1 \quad \cdots \quad a_1 + b_0 \quad \cdots \quad a_{m-1} + b_{n-1}]$$

with  $a_i$  being the position of the  $i^{\text{th}}$  set element of  $\mathbf{A}$  and  $b_j$  being the position of the  $j^{\text{th}}$  set element of  $\mathbf{B}$ . The result is the list of set positions in the product, with all sums being performed modulo  $p$  to take into account the rotation of set bits “out of the right margin and back into the left one”. There is one more problem, however, that is the cancellation of terms: if both  $\mathbf{A}$  and  $\mathbf{B}$  have set positions  $[0, 1]$  the product will have set positions  $[0, 2]$ , but this algorithm will output  $[0, 1, 1, 2]$ . It is then necessary to eliminate duplicates that appear an even number of times: this implies the actual weight of the result is unknown. Given the added complexity of tracking weights has no real advantage in terms of memory usage, as the space reserved for each operation must be the maximum one possibly occupied by the result, it was chosen to simply fill the unused position with illegal values.

The hardware implementation is as follows:

- get  $a_0$  and  $b_0$ , then compute  $a_0 + b_0$  and save the result modulo  $p$  in a temporary variable
- get  $b_1$ , compute  $a_0 + b_1$ , save the result modulo  $p$  in a temporary variable
- continue until all combinations have been processed, we now have a temporary result in memory
- sort the temporary result to have all set positions in order: this is done in place with an insertion sort algorithm[2] that does not require additional space in memory
- go through the temporary result and copy all values that are different from the one immediately following them in the memory space for the real result: if two successive values are equal skip them both
- if it was not skipped as a result of the previous value, copy the last value of the temporary result into the real result (the previous iteration requires a “next value” to compare to and can’t be applied to the last element)
- fill all remaining memory location assigned to the result with “invalid” flags, i.e. illegal values: in line with the software implementation,  $p$  was used as invalid flag (since the last legal position is  $p - 1$ )

### 4.1.1 Out-of-order result and modulo $p$ implementation

The implementation of anything having to do with division is usually very costly in terms of area and performance, either taking multiple cycles to compute a result or needing very big combinational networks to compute the result. The particular problem at hand does once again provide a way to reduce complexity, allowing for a short critical path using little area:

$$a_i < p$$

$$b_j < p$$

$$a_i + b_j < 2p - 1$$

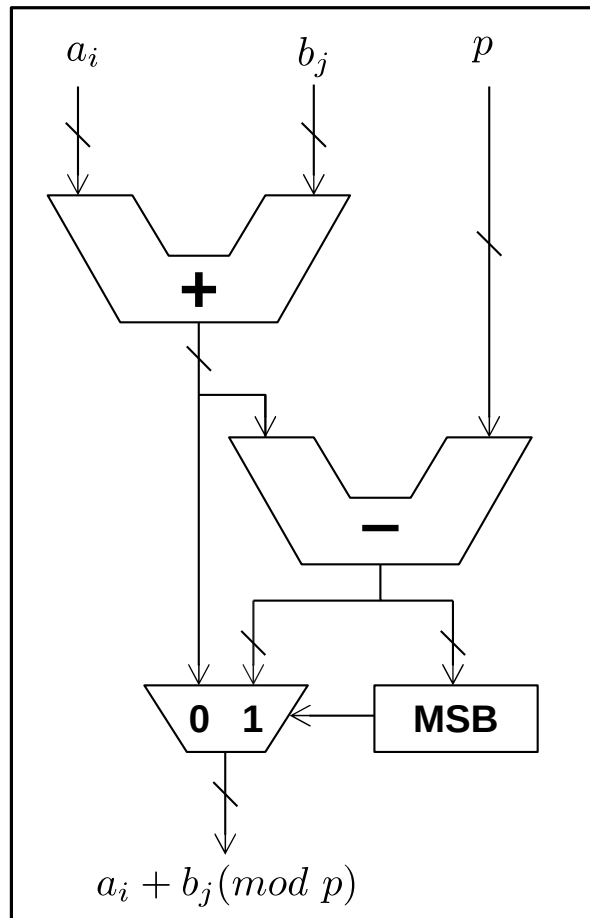
This means that there are only two possible solutions to  $a_i + b_j \pmod{p}$ : either  $a_i + b_j$  or  $a_i + b_j - p$ . The second value is computed in the same cycle as the first one and a simple comparator then selects the proper result: this is easily done by checking the sign of  $a_i + b_j - p$ , as that is the correct result if it is positive (a circuit performing the operation is shown in figure 4.1).

The state machine (depicted in figure 4.2) controlling the actual operation is actually quite simple, depending mostly on parameters known at compile time: it sits in an idle state until a “start” signal is received, at which point two nested loops are performed to select all combinations of  $a_i$  and  $b_j$ . Values  $i$  and  $j$  are added as offsets to the base addresses of the two circulant blocks, provided by the parent module that controls the multiplication of  $\mathbf{Q}^T$  by  $\mathbf{H}^T$ . The address of the result  $z_k$  is obtained by summing to the base address of the result block a counter  $k$ , incremented in the inner loop and reset when the block is idle. Once the result is ready in memory, the state machine raises a flag and stays in a “done” state until the “start” signal is deasserted: it then moves to an idle state and can perform another multiplication.

The implementation computes one result per cycle and writes it immediately to memory, then it moves to the next value on the following cycle.

If a real implementation suffers from critical path problems while trying to achieve this, since the circulant multiplication block has no feedback, it can be pipelined without side effects as long as the frequency of the memory can keep up with the frequency of the decoder itself, at which point the memory-bound nature of the problem requires reconsidering how data are stored.

Storing data in multiple memories is possible, interleaving access to each of them and thus multiplying the effective maximum frequency of the decoder at the cost of more buses: this is easily done by using the least significant bits of the addresses as computed by the existing modules as inputs to a decoder



sum and modulo

Figure 4.1: Encryption in LEDAcrypt

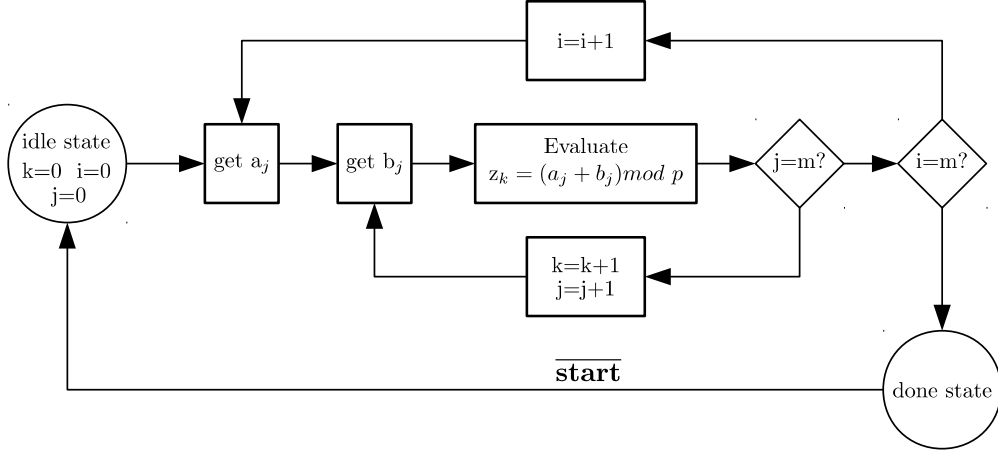


Figure 4.2: Multiplication between two circulant blocks

for memory selection, thus cycling through all memories before coming back to the initial one. A future modification would however need a thorough investigation on the consequence of such a choice on the system at large, to ensure all modules do have sequential access to memory locations: if this assumption does not hold additional logic is required to slow down the decoder when there is the danger of multiple subsequent accesses to the same memory.

### 4.1.2 Result sorting

The result as computed up to now is out-of-order, meaning that the list of positions of set bits is not increasing: this is not technically a problem in terms of end result, but efficient implementation of some operations require ensuring that the list is monotonically increasing. As such, a sorting step is needed: the insertion sort was chosen due to the algorithm simplicity (directly translating to hardware complexity and ease of implementation) and because it is an “in place” algorithm that does not require additional memory other than a temporary variable to swap adjacent values.

The insertion sort is based on two nested loops, the outer one moving from the start of the list to its end and the inner one moving back until the correct position for the element pointed in the outer loop is found. The algorithm performs the following operations:

- from the starting list two lists are built: the first, sorted, is initially made of the first element of the starting list; the second one is all the rest

- the first element of the unordered list is compared to the last element of the ordered list
- if the new element is bigger than the biggest element of the ordered list, it is appended at the end of the ordered list; if it is not, the biggest element is moved to the end of the list (now one position “right”) and the new element is compared with the second biggest one
- comparisons continue until the new element is bigger than the old one we are comparing it to or the beginning of the list is reached, then the new element is placed just after the one it was compared against
- a new element is taken from the unordered list and the previous steps are repeated until all elements are moved to the sorted list

The hardware implementation is extremely simple, consisting of a comparator and a register containing the value being inserted in the current iteration. The value from the sorted sub-list is directly taken from memory, and the smaller of the two is selected by a multiplexer and sent back to memory in the position right after the one in which the already-sorted element resides. A simple control unit takes care of selecting a new element to insert (outer loop, increasing a counter each time the previous element is inserted) and selecting the appropriate values to compare this element against (inner loop, decreasing a second counter that starts one off the current value of the first one).

Future improvements to the sorting operation could come from the study of an ad-hoc algorithm tailored to the specific distribution of the out-of-order result, possibly taking advantage of the monotonic segments that are already present to implement a custom merge-sort. No additional investigation was done in this direction.

### 4.1.3 Modulo 2 on compressed matrices

Given a circulant block stored in memory as defined in previous sections, any position containing a value  $n$  is present  $n$  times in the list of set positions. Due to the result being sorted, any position containing a value bigger than once will be present multiple times in adjacent positions in memory, thus allowing for a fast elimination of pairs in a single pass. The elimination of pairs results in positions appearing an odd number of times reduced to appearing only once and positions appearing an even number of times disappearing completely, thus getting a modulo 2 multiplication from the partial result over the natural numbers.

The hardware implementation uses two counters to cycle through memory: the first one is used to access two adjacent memory cells to compare their content (an actual synthesis might prefer to have two separate counters offset by 1 and avoid the combinational logic needed to compute the increment), the second one points at the cell where the value is going to be copied. The algorithm is as follows:

- Set  $i, j$  to 0 ( $i$  and  $j$  are offsets from the base of the list in memory)
- Get the  $i^{\text{th}}$  and the  $(i+1)^{\text{th}}$  elements of the list, from here on  $a$  and  $b$
- If  $a \neq b$  copy  $a$  in place of the  $j^{\text{th}}$  element of the list, increase  $i$  and  $j$ ; if  $a = b$  increase  $i$  twice
- Repeat until  $i$  points either to the last element of the list or to the memory cell just after
- If  $i$  points to the last element of the list copy it in place of the  $j^{\text{th}}$  element of the list
- Fill the rest of the list with data recognizable as invalid, such as  $p$

## 4.2 Circulant block sum

Summing two circulant matrices stored in the format in use is simply done by concatenating the list of set positions and then taking the modulo 2 like it is done with the multiplication. A more efficient approach is however possible by merging the sorting and the modulo operation with the concatenation, in order to avoid doing these necessary steps later on.

This is done with three different counters used to point an element of the first block, an element of the second and the cell where the result will be stored. The two operand positions are compared: they get discarded if they are the same (this ensures the result is modulo 2 without additional operations), otherwise the smaller one is copied in the result cell and the next position from its block is fetched for the next cycle. The precedence in copying the smaller position first results in the sum being ordered.

The exact algorithm is as follows:

- Set  $i, j, k$  to 0 (offsets from the base of the lists containing the set positions of the first operand, the second operand and the result)



- Retrieve the positions pointed by  $i$  and  $j$  (from here on  $a$  and  $b$ ) and compare them: if  $a = b$  increment  $i$  and  $j$ , if  $a < b$  copy  $a$  in the memory cell pointed by  $k$  and increment  $i$  and  $k$ , if  $a > b$  copy  $a$  in the memory cell pointed by  $k$  and increment  $j$  and  $k$
- Repeat until  $a$  and  $b$  are either invalid or all values have been processed
- Fill all remaining positions of the result (if any) with invalid values

#### 4.2.1 Memory movement

The sum of two circulant blocks as shown above is not done in-place, as there is no way to ensure that no information that is still needed would not be overwritten by the ongoing operation. As such, it is needed to move the result from a temporary location to its final destination. The hardware performing this operation is extremely simple and uses a single counter as offset to two different base positions in memory, copying the values from the first into the second until done.

### 4.3 Quasi-cyclic multiplication

The aforementioned submodules implement operations among circulant blocks, which are however not what we are interested in per se: the objective of the key reconstruction operation is getting  $\mathbf{L}^T$ , which is not a single circulant block. As such, we need to show that operations among quasi-cyclic matrices can be expressed as operations on their single circulant blocks.

Given that  $\mathbf{L}^T = \mathbf{Q}^T \mathbf{H}^T$ , the standard implementation of matrix multiplication would consist in computing the sum of the element-wise multiplication between the  $i^{\text{th}}$  row of  $\mathbf{Q}^T$  and the  $j^{\text{th}}$  column of  $\mathbf{H}^T$  to obtain each element  $l_{i,j} \in \mathbf{L}^T$ :

$$\mathbf{Q}^T = \begin{bmatrix} \mathbf{Q}_{0,0} & \mathbf{Q}_{0,1} & \cdots & \mathbf{Q}_{0,n_0-1} \\ \mathbf{Q}_{1,0} & \mathbf{Q}_{1,1} & \cdots & \mathbf{Q}_{1,n_0-1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{Q}_{n_0-1,0} & \mathbf{Q}_{n_0-1,1} & \cdots & \mathbf{Q}_{n_0-1,n_0-1} \end{bmatrix} = \begin{bmatrix} q_{0,0} & q_{0,1} & \cdots & q_{0,n_0p-1} \\ q_{1,0} & q_{1,1} & \cdots & q_{1,n_0p-1} \\ \vdots & \vdots & \ddots & \vdots \\ q_{n_0p-1,0} & q_{n_0p-1,1} & \cdots & q_{n_0p-1,n_0p-1} \end{bmatrix}$$

$$\mathbf{H}^T = \begin{bmatrix} \mathbf{H}_0 \\ \mathbf{H}_1 \\ \vdots \\ \mathbf{H}_{n_0-1} \end{bmatrix} = \begin{bmatrix} h_{0,0} & h_{0,1} & \cdots & h_{0,p-1} \\ h_{1,0} & h_{1,1} & \cdots & h_{1,p-1} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n_0p-1,0} & h_{n_0p-1,1} & \cdots & h_{n_0p-1,p-1} \end{bmatrix}$$

$$\mathbf{L}^T = \begin{bmatrix} \mathbf{L}_0 \\ \mathbf{L}_1 \\ \vdots \\ \mathbf{L}_{n_0-1} \end{bmatrix} = \begin{bmatrix} l_{0,0} & l_{0,1} & \cdots & l_{0,p-1} \\ l_{1,0} & l_{1,1} & \cdots & l_{1,p-1} \\ \vdots & \vdots & \ddots & \vdots \\ l_{n_0p-1,0} & l_{n_0p-1,1} & \cdots & l_{n_0p-1,p-1} \end{bmatrix}$$

$$l_{i,j} = \sum_{n=0}^{n_0p-1} q_{i,n} h_{n,j}$$

From the previous equation, simple algebra shows that:

$$l_{i,j} = \sum_{m=0}^{n_0-1} \left( \sum_{n=mp}^{(m+1)p-1} q_{i,n} h_{n,j} \right)$$

While the latter equation is somewhat inelegant, it expresses an important property of the system at hand: it is possible to break up the sum to work with smaller vectors (in our case of size  $p$ ) without affecting the final result. This is important because multiplying the circulant block  $\mathbf{Q}_{x,m}$  by the circulant block  $\mathbf{H}_m$  results in:

$$\lambda_{i \pmod{p},j} = \sum_{n=mp}^{(m+1)p-1} (q_{i,n} h_{n,j}) \quad i \in [xp, (x+1)p-1]$$

with  $\lambda_{i \pmod{p},j}$  being the element of  $\mathbf{Q}_{x,m}\mathbf{H}_m$  in row  $i \pmod{p}$  and column  $j$ . It is then possible to expand on this result with:

$$\sum_{m=0}^{n_0-1} (\mathbf{Q}_{x,m}\mathbf{H}_m) = \sum_{m=0}^{n_0-1} \left( \sum_{n=mp}^{(m+1)p-1} q_{i,n} h_{n,j} \right) \quad i \in [xp, (x+1)p-1]$$

which is computing all  $l_{i,j} : i \in [xp, (x+1)p-1]$  in parallel, using the extremely efficient implementation allowed by the representation of circulant blocks. It can then be noted that:

$$\sum_{m=0}^{n_0-1} (\mathbf{Q}_{x,m}\mathbf{H}_m) = \mathbf{L}_x$$

Iterating through  $x \in [0, n_0-1]$  it is then possible to obtain the full  $\mathbf{L}^T$  matrix only using multiplication and sum of circulant blocks and concatenating the results.

The hardware implementation uses a request-based system in which each module implementing an operation over circulant blocks is inactive until

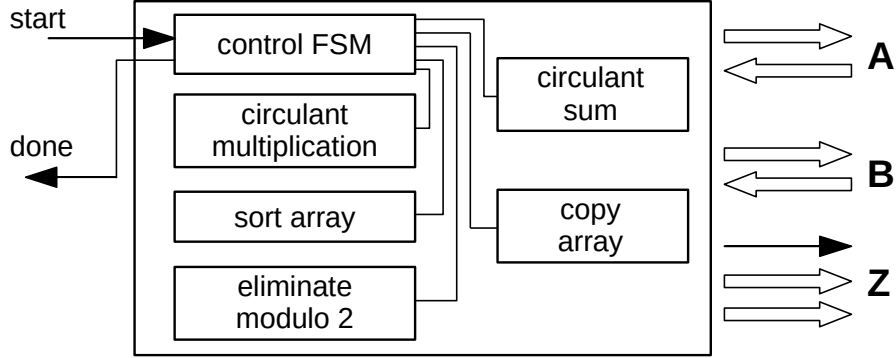


Figure 4.3: Module computing  $\mathbf{L}^T = \mathbf{Q}^T \mathbf{H}^T$

explicitly awoken by the control unit. Each module has moreover as inputs the base position in memory of the circulant blocks on which it will perform the operation (both operands and result, where applicable) and the maximum size of the inputs, needed because the weight of  $\mathbf{Q}_{x,m}$  depends on  $x$  and  $m$ . The outputs to memory of each submodule are multiplexed by the control unit and forwarded up the hierarchy, while the “operation done” signals are used to move between states of a finite state machine without the need to know the exact duration of the operation beforehand.

The state machine is as follows:

- set  $i, j$  to 0 ( $i, j$  indexes of circulant blocks in  $\mathbf{Q}^T$  and  $\mathbf{H}^T$ )
- when instructed to start, multiply  $\mathbf{Q}_{i,j}$  by  $\mathbf{H}_j$  and put the result in a temporary location
- sort the result in place
- get the result in modulo 2 in place
- if  $j = 0$  copy the result into  $\mathbf{L}_i$ , else sum the result to  $\mathbf{L}_i$  and save the sum in a temporary location
- copy the sum to  $\mathbf{L}_i$
- repeat for all  $j$ , then set  $j = 0$  and repeat for all  $i$
- signal that the operation is done

# Chapter 5

## Vector by matrix multiplication (and vice-versa)

After retrieving  $\mathbf{L}^T$ , all information needed to compute the syndrome of the received code  $\underline{x}$  is available. The syndrome is computed as

$$\underline{s} = \underline{x}\mathbf{L}^T$$

The needed operation is the multiplication of a vector by a matrix, which is recurrent in the main decoding loop too: it is thus paramount to get a performant module that can be time multiplexed and that is flexible enough to be capable of handling different sizes of vectors and matrices.

### 5.1 Vector by circulant matrix

As all matrices involved in the decoding operation are concatenations of circulant blocks, the most basic building block would be a module capable of multiplying a vector having length  $p$  by a circulant block or vice-versa: the proof that this is sufficient is deferred to individual sections below.

For what concerns all operations in this section, it is assumed that  $\underline{a}$  is a vector of length  $p$  stored in memory as a concatenation of individual values (that might or might not be binary) and  $\mathbf{B}$  is a binary circulant block of weight  $w$  stored in the usual “set positions” format. Vector  $\underline{c}^T$  is the result of  $\underline{a}^T\mathbf{B}$  and has size  $p$  too.

$$\underline{a}^T = [a_0 \quad a_1 \quad \cdots \quad a_{p-1}]$$

$$\mathbf{B} = \begin{bmatrix} b_0 & b_1 & \cdots & b_{p-1} \\ b_{p-1} & b_0 & \cdots & b_{p-2} \\ \vdots & \vdots & \ddots & \vdots \\ b_1 & b_2 & \cdots & b_0 \end{bmatrix}$$

$$\underline{c}^T = [c_0 \quad c_1 \quad \cdots \quad c_{p-1}]$$

It can be noted that  $c_0$  is, trivially:

$$c_0 = a_0 b_0 + a_1 b_{p-1} + \cdots + a_{p-1} b_1$$

More interestingly, this same relation can be expressed as:

$$c_0 = \sum_{n=0}^{p-1} a_n b_{p-n}$$

Similarly:

$$c_1 = a_0 b_1 + a_1 b_0 + \cdots + a_{p-1} b_2$$

$$c_1 = \sum_{n=0}^{p-1} a_n b_{p-n+1} \pmod{p}$$

This is finally expanded into a single relation that states:

$$c_i = \sum_{n=0}^{p-1} a_n b_{p-n+i} \pmod{p}$$

We thus have a universal analytic expression for the value of any  $c_i$  given that  $\mathbf{B}$  is circulant. Due to the sparsity of  $\mathbf{B}$  (we remind that  $\mathbf{B}$  is a block of  $\mathbf{H}^T$ ,  $\mathbf{Q}^T$  or  $\mathbf{L}^T$ ) it is possible to entirely avoid operations in which  $b_m$  is not set, saving a lot of time.

To further optimize the hardware implementation of the operation, due to the sum involving three operands (the accumulator,  $a_n$  and  $b_m$ ) while we only assumed two read ports were available,  $b_m$  (actually  $m$  itself, given the way matrices are stored) is read first and stored in a register, and all operations involving that single  $b_m$  are computed before moving to the next one. This is more efficient than reading  $a_n$  and storing that, as  $p$  operations involve  $b_m$  while only  $w$  operations involve  $a_n$ : reading the operands the other way around results in  $p - w$  wasted cycles. A loop over  $n$  retrieves all  $a_n$  and points the affected result  $c_{m+n \pmod{p}}$ , then the next set  $m$  (easily found in the next position in memory) is retrieved and stored and the operation is repeated until the last  $m$  is reached, at the  $w^{\text{th}}$  iteration.

At that point the result is ready and the module signals that the operation is finished.

## 5.2 Circulant matrix by vector

We now analyze the case in which  $\underline{d} = \mathbf{B}\underline{a}$ :

$$\underline{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{p-1} \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} b_0 & b_1 & \cdots & b_{p-1} \\ b_{p-1} & b_0 & \cdots & b_{p-2} \\ \vdots & \vdots & \ddots & \vdots \\ b_1 & b_2 & \cdots & b_0 \end{bmatrix}$$

$$\underline{d} = \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{p-1} \end{bmatrix}$$

It can be trivially obtained that:

$$d_0 = \sum_{n=0}^{p-1} b_n a_n$$

$$d_1 = \sum_{n=0}^{p-1} b_{n-1} \pmod{p} a_n$$

And the result can be generalized to:

$$d_i = \sum_{n=0}^{p-1} b_{n-i} \pmod{p} a_n$$

This result is very similar to what we obtained in the previous section. Indeed, we can write the two results such that:

$$c_i = \sum_{n=0}^{p-1} a_n b_{(i-n) \pmod{p}}$$

$$d_i = \sum_{n=0}^{p-1} a_n b_{-(i-n) \pmod{p}}$$

The hardware that controls the two operations can thus be the same, fixing  $m$  (index of  $b_m$ ) and sweeping through  $n$  to obtain  $i$ . The insertion of a simple control signal lets us select whether we want to perform  $\underline{a}\mathbf{B}$ , in which case we compute the result address as  $i = n + m \pmod{p}$ , or  $\mathbf{B}\underline{a}$ , in which case the result address is  $i = n - m \pmod{p}$ .

The only modification needed to support both operations is thus using an adder-subtractor instead of a simple adder in the target address computation section: the entirety of the control finite state machine is shared.

### 5.3 $\underline{x}$ by $\mathbf{L}^T$

The starting syndrome of the code is:

$$\underline{s} = \underline{x}\mathbf{L}^T$$

$$\underline{s} = [s_0 \quad s_1 \quad \cdots \quad s_{p-1}]$$

We can write  $\underline{x}$  as:

$$\underline{x} = [\underline{x}_0 \quad \underline{x}_1 \quad \cdots \quad \underline{x}_{n_0-1}] = [x_0 \quad x_1 \quad \cdots \quad x_{p-1} \quad x_p \quad \cdots \quad x_{n_0p-1}]$$

$\underline{x}$  is thus split into  $n_0$   $p$ -length vectors, while  $\mathbf{L}^T$  is by construction split in blocks already.

$$\mathbf{L}^T = \begin{bmatrix} \mathbf{L}_0 \\ \mathbf{L}_1 \\ \vdots \\ \mathbf{L}_{n_0-1} \end{bmatrix}$$

By definition,  $s_i$  is the sum of the element-wise multiplication between  $\underline{x}$  and the  $i^{\text{th}}$  column of  $\mathbf{L}^T$ . We hereby define  $\underline{s}_k$  as:

$$\underline{s}_k = \underline{x}_k \mathbf{L}_k$$

Each of such  $\underline{s}_k$  is thus a partial sum and we can get then  $\underline{s}$  as:

$$\underline{s} = \sum_{k=0}^{n_0-1} \underline{s}_k$$

It is thus possible to obtain  $\underline{s}$  through multiplications of a vector by a circulant matrix, using the module we described in the previous section. Due to the particular implementation of the module, moreover, all multiplications behave as a “multiply and accumulate” operation, meaning there is no need to actually implement the sum thus saving area and execution time.

The module performing this operation is thus simply a control unit that provides the base address in memory of the appropriate slice of  $\underline{x}$  and of the proper block of  $\mathbf{L}^T$  (the latter of which is somewhat complicated by the fact that different blocks of  $\mathbf{L}^T$  have different weight, but is resolved with a simple look-up table). The unit then instructs the vector-by-circulant core to perform a vector by matrix multiplication, storing the result in the base address of  $\underline{s}$ , and waits for the multiplication core to return to then provide new values for the base addresses of the operands. Once the  $n_0^{\text{th}}$  multiplication has returned the control unit itself returns.

## 5.4 $\mathbf{H}^T$ by $\underline{s}^T$

In the main decoding loop the first operation is obtaining  $\underline{\Sigma}$  as:

$$\left(\underline{\Sigma}^{(l)}\right)^T = \mathbf{H}^T \left(\underline{s}^{(l-1)}\right)^T$$

$$\left(\underline{\Sigma}^{(l)}\right)^T = \begin{bmatrix} \sigma_0 \\ \sigma_1 \\ \vdots \\ \sigma_{n_0 p - 1} \end{bmatrix}$$

Given that  $\mathbf{H}^T$  is:

$$\mathbf{H}^T = \begin{bmatrix} \mathbf{H}_0 \\ \mathbf{H}_1 \\ \vdots \\ \mathbf{H}_{n_0 - 1} \end{bmatrix}$$

and  $\underline{s}^T$  has  $p$  elements, each  $\sigma_i$  can be obtained by multiplying a row of a single block by  $\underline{s}^T$ . The result will then be not the sum of many terms like before, but the concatenation: this is simply done incrementing the base position of the matrix-by-vector result by  $p$  between operations.

$$\left(\underline{\Sigma}^{(l)}\right)^T = \begin{bmatrix} \underline{\Sigma}_0 \\ \underline{\Sigma}_1 \\ \vdots \\ \underline{\Sigma}_{n_0 - 1} \end{bmatrix}$$

$$\underline{\Sigma}_i = \mathbf{H}_i \underline{s}^{(l-1)}$$

The hardware implementation is similar to the one used previously, but provides a new base address for the result of each multiplication instead



of accumulating over the same one and instructs the multiplication core to perform a circulant-by-vector operation.

## 5.5 $\mathbf{Q}^T$ by $\underline{\Sigma}^T$

After obtaining  $\underline{\Sigma}$ ,  $\underline{R}$  is obtained as:

$$\underline{R}^{T(l)} = \mathbf{Q}^T \underline{\Sigma}^{T(l)}$$

$$\underline{R}^{T(l)} = \begin{bmatrix} r_0 \\ r_1 \\ \vdots \\ r_{n_0 p - 1} \end{bmatrix}$$

The operation is more complex as  $\mathbf{Q}^T$  is square, thus both concatenation and sum will be needed:

$$\mathbf{Q}^T = \begin{bmatrix} \mathbf{Q}_{0,0} & | & \mathbf{Q}_{0,1} & | & \cdots & | & \mathbf{Q}_{0,n_0-1} \\ \mathbf{Q}_{1,0} & | & \mathbf{Q}_{1,1} & | & \cdots & | & \mathbf{Q}_{1,n_0-1} \\ \vdots & | & \vdots & | & \ddots & | & \vdots \\ \mathbf{Q}_{n_0-1,0} & | & \mathbf{Q}_{n_0-1,1} & | & \cdots & | & \mathbf{Q}_{n_0-1,n_0-1} \end{bmatrix}$$

$$\underline{R}^T = \begin{bmatrix} \underline{R}_0 \\ \underline{R}_1 \\ \vdots \\ \underline{R}_{n_0-1} \end{bmatrix}$$

$$\underline{R}_i = \sum_{j=0}^{n_0-1} \mathbf{Q}_{i,j} \underline{\Sigma}_j$$

The implementation is more complicated than the other ones, but it keeps the same basic principle: two nested loops iterate over  $j$  and  $i$  providing the base addresses of  $\underline{R}_i$ ,  $\mathbf{Q}_{i,j}$  and  $\underline{\Sigma}_j$ , with the appropriate values for the base of circulant blocks provided by a look-up table

## 5.6 $\underline{e}$ by $\mathbf{H}^T$

The last operation in the decoding loop involves computing an increment vector for the syndrome, as

$$\underline{\Delta s}^{(l)} = \underline{e}^{(l)} \mathbf{H}^T$$

Since  $\underline{e}$  has the same size as  $\underline{x}$  and  $\mathbf{H}$  has the same size as  $\mathbf{L}$ , this operation is exactly equivalent to the multiplication  $\underline{s} = \underline{x}\mathbf{L}^T$ . Again, due to the multiplication really behaving as a “multiply and accumulate”, the result  $\underline{\Delta s}$  is directly summed to  $\underline{s}$  with no overhead.

# Chapter 6

## Error update

Vector  $\underline{R}$  contains the count of unsatisfied parity checks in which the corresponding bit of  $\underline{x}$  is involved. To proceed with the algorithm, the bits which are most likely to be wrong need to be found.

### 6.1 Peak search

Finding the peaks of  $\underline{R}$  is done via a simple single-pass sequential algorithm, although it would be possible to parallelize the algorithm by replicating the hardware and adding logic to merge the results. Still, the time consumption of this step is sufficiently small with respect to the total required for the loop (dominated by vector-by-matrix multiplications) that this parallelization was deemed unnecessary for this experimental implementation.

The hardware implementation consists of a temporary register containing the current max and an array of fixed, arbitrary size to contain the position of all values equal to the current max. While the size of the array can be changed, having it too small will impact performance and possibly the stability of the algorithm, while having it too big will result in a very high area footprint. The maximum is initialized to 0 and the array is initialized to all invalid positions, then each time a value equal to the maximum is found its position is appended to the array, if there is space left. If the array is full, no operation is performed and the algorithm continues normally. Each time a value greater than the maximum is found, the maximum is updated, the array is flushed and the first value of the array is set to the position of the new maximum. The algorithm then completes once the entirety of  $\underline{R}$  has been walked through, and returns the array for usage in the next module.

## 6.2 Row extraction from compressed matrix

The next step in the algorithm requires summing to the current error  $\underline{e}$  the rows of  $\mathbf{Q}^T$  having the index of the found maxima. As we do not have the matrix stored in a readily-available format for this operation (we only have the first row of each block, while we need individual row access), a relation between the set positions in the first row of each module and the set positions in an arbitrary row must be found.

One complication is that any row  $\underline{q}_k$  stretches over multiple blocks  $\mathbf{Q}_{i,j}$ :

$$\mathbf{Q}^T = \begin{bmatrix} \mathbf{Q}_{0,0} & \mathbf{Q}_{0,1} & \cdots & \mathbf{Q}_{0,n_0-1} \\ \mathbf{Q}_{1,0} & \mathbf{Q}_{1,1} & \cdots & \mathbf{Q}_{1,n_0-1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{Q}_{n_0-1,0} & \mathbf{Q}_{n_0-1,1} & \cdots & \mathbf{Q}_{n_0-1,n_0-1} \end{bmatrix} =$$

$$= \begin{bmatrix} q_{0,0} & q_{0,1} & \cdots & q_{0,p-1} & q_{0,p} & \cdots & q_{0,n_0p-1} \\ q_{1,0} & q_{1,1} & \cdots & q_{1,p-1} & q_{1,p} & \cdots & q_{1,n_0p-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ q_{p-1,0} & q_{p-1,1} & \cdots & q_{p-1,p-1} & q_{p-1,p} & \cdots & q_{p-1,n_0p-1} \\ q_{p,0} & q_{p,1} & \cdots & q_{p,p-1} & q_{p,p} & \cdots & q_{p,n_0p-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ q_{n_0p-1,0} & q_{n_0p-1,1} & \cdots & q_{n_0p-1,p-1} & q_{n_0p-1,p} & \cdots & q_{n_0p-1,n_0p-1} \end{bmatrix}$$

We can get the  $l^{\text{th}}$  row of a circulant block  $\mathbf{A}$ :

$$\mathbf{A} \begin{bmatrix} a_0 & a_1 & \cdots & a_{p-1} \\ a_{p-1} & a_0 & \cdots & a_{p-2} \\ \vdots & \vdots & \ddots & \vdots \\ a_1 & a_2 & \cdots & a_0 \end{bmatrix}$$

$$\underline{a}_l^T = [a_{p-l \pmod{p}} \quad a_{p-l+1 \pmod{p}} \quad a_{p-l+2 \pmod{p}} \quad \cdots \quad a_{2p-l-1 \pmod{p}} \quad (\pmod{p})]$$

This means it is possible to get any  $\underline{q}_k$  as concatenation of rows of the appropriate blocks. The blocks involved are all blocks  $\mathbf{Q}_{i,j}$  with  $i = \text{floor}(k/p)$ , while  $l$  is obtained as  $l = i \pmod{p}$ .

## 6.3 Vector plus compressed row

Due to the blocks  $\mathbf{Q}_{j,k}$  being stored in compressed format and the sparsity of the rows, it is convenient to handle  $\underline{q}_k$  in  $n_0$  chunks of length  $p$  and maintain

the compressed format on the result, in order to have a list of set positions matching the positions that will need to be flipped in the corresponding chunks of  $\underline{e}$ . This is easily done reading all  $l$  corresponding to set bits in the first row of the block and applying the operation we described in the previous section, with the result being the list of set positions for  $\underline{q}_k$ .

The actual sum consists in computing  $i$  and  $l$  in order to get the affected row of blocks and individual row offset, then iterating through the row of blocks one at a time to compute the list of set bits and flipping the corresponding bits in  $\underline{e}$ . Once done with a row, the next  $k$  is fetched from the list of rows to be summed and the operation is repeated until either all rows have been summed or  $k$  is invalid, indicating that the number of peaks in  $\underline{R}$  was smaller than the maximum supported by the decoder.

The operation is done incrementally in place, so that no additional memory is needed to store intermediate results.

# Chapter 7

## Main loop state machine

The entirety of the decoder is controlled by the top-level state machine, calling the various functions as they are needed according to the algorithm. This state machine is the bus arbitrator that multiplexes the RAM signals coming from the various blocks and forwards them to the external pins.

The state machine performs the following operations:

- wait for the “start” signal
- ask the key reconstruction module to retrieve  $\mathbf{L}^T$
- ask the module performing the  $\underline{x}\mathbf{L}^T$  operation to compute the syndrome  $\underline{s}$
- ask the module performing the  $\mathbf{H}^T \underline{s}^T$  operation to compute  $\underline{\Sigma}^T$
- ask the module performing the  $\mathbf{Q}^T \underline{\Sigma}^T$  operation to compute  $\underline{R}^T$
- ask the peak finder module to compute the positions of the maxima of  $\underline{R}$
- ask the sum module to add all the rows of  $\mathbf{Q}^T$  corresponding to the found maxima to  $\underline{e}$
- ask the module performing the  $\underline{e}\mathbf{H}^T$  operation to update  $\underline{s}$
- clear all temporary results, including  $\underline{\Sigma}$  and  $\underline{R}$
- check whether  $\underline{s}$  is null or the iteration limit was reached: if one of the conditions holds compute the message, otherwise loop back to the step that computes  $\underline{\Sigma}$  and increase the iteration counter

- return the “done” signal; in case the iteration limit was reached, report a decoding failure

As multiplication operations are implemented as “multiply and accumulate”, results from previous iterations would add up continuously. While this is desirable for certain vectors ( $\underline{s}$  and  $\underline{e}$ ), it is disruptive for all the others: as such a memory clear step is performed between iterations zeroing the memory sections containing  $\underline{\Sigma}$  and  $\underline{R}$

## 7.1 Design modularity and shared resources

Deep emphasis was put in the reutilization of the same basic blocks over and over in order to save resources, wherever this was possible. The only common blocks that could potentially be shared and were not arranged to be are the mod  $p$  operators: this was a deliberate choice to ease code development and readability, while the module itself is reasonably simple so that the area overhead is not too high. In terms of actual implementation, this maps to more raw silicon needed for the gates but less routing and no multiplexing.

Modularity was achieved through a “function call” architecture that was devised to make each module accept any data that fit with the template, that being either a vector of length  $p$  or a circulant block and its weight. All vectors and blocks are passed by reference as pointers to memory, so that actual data transfer between block is minimal. The algorithm implementing the decoder is not suited to pipelining, but this very drawback is what allows for the resource sharing as all operations performed at different points of the algorithm are ensured not to be called concurrently.

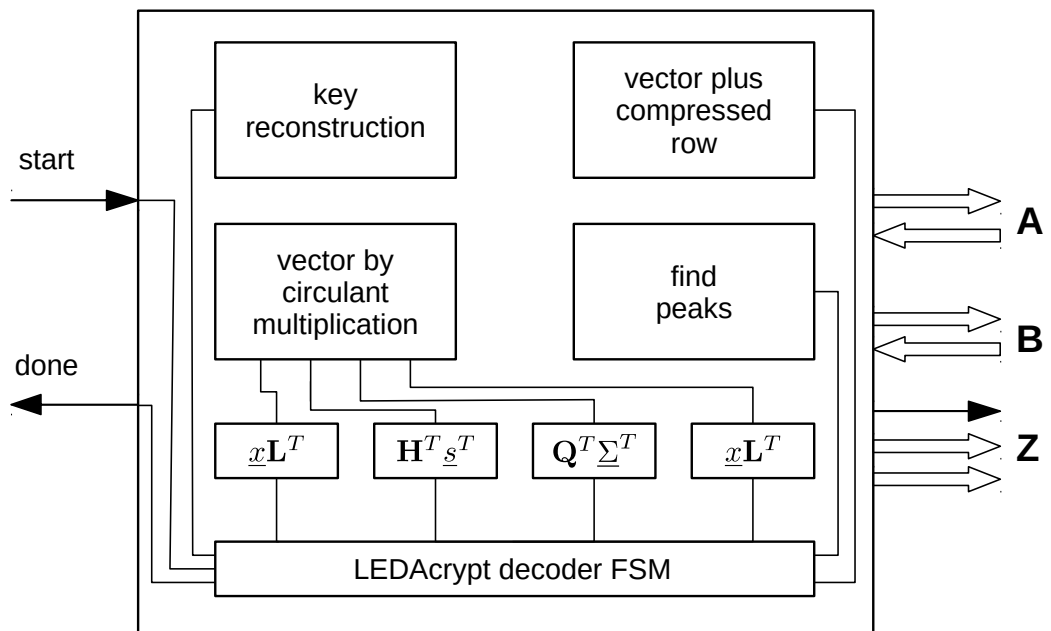


Figure 7.1: Decryption module



# Chapter 8

## Conclusions

The proposed implementation is but a first step in the study of the feasibility on silicon of cryptographic QC-LDPC codes. While all operations needed as basic blocks to decode the input are simple, well known and efficient, the architecture at large is very much experimental and might present severe bottlenecks in high-frequency operation, especially on the side of data transfers to memory which are paramount to the decoder.

Future improvements are likely to come in the form of an additional memory management layer, translating requests to a complex memory structure able to maximize the data rate. This could be done in much the same way as was devised for hard disks with the RAID architecture[9], with multiple separated memory units having independent access that would thus be able to transfer, albeit at slow speed for each transaction, a massive amount of data per cycle. Additionally, while having memory internal to the decoding unit itself would be unfeasible for vectors (all of which have length at least  $p$ ), the cost of storing internally the compressed  $\mathbf{H}^T$ ,  $\mathbf{Q}^T$  and  $\mathbf{L}^T$  matrices is quite low: this makes it possible to have a fast portion of memory that is expected to be accessed in a single cycle even at very high frequencies.

Minor improvements in terms of resource sharing can be gained by unifying the control finite state machines performing the multiplications  $\underline{s} = \underline{x}\mathbf{L}^T$  and  $\underline{\Delta s} = \underline{e}\mathbf{H}^T$  and possibly sharing a single modulo  $p$  computation unit.

In terms of actual algorithm parallelization and assuming the problem of the memory bottleneck as completely solved, the computation of the unordered result  $\mathbf{L}^T = \mathbf{Q}^T\mathbf{H}^T$  can be performed in parallel by simple replication of the processing unit, with the limit being computing all its element in one single pass. Investing bigger area than currently allotted it would also be possible to use faster sorting algorithms like the merge sort[10], while elimination of adjacent doubles from a list to implement the modulo 2 and the sum of matrices could be done with a two-cycle operation operating first on

even-odd pairs and then on odd-even ones. Still, all of this only results in speeding up the key reconstruction which happens only once.

Parallelizing operations involving vectors is more challenging, due to the sheer size of the vectors themselves. Throughout chapter 5 it was shown that all operations on vectors can be reduced to operations on length  $p$  vectors, but  $p$  is very big nonetheless. Multiplications with circulant blocks are in essence circular shifts and sums: a system to implement shifts over sections of a vector (as opposed to the entirety of it) can be obtained by simply having multiple units performing the operation. Peak finding can be carried out on segments and the results merged. The row sum operation can be carried out in parallel for each row, although the benefit of doing so is likely minimal.

# Appendix A

## Source code

### A.1 Key reconstruction

#### Circulant multiplication

```
1
2 — Author: Flavio Tanese
3 — Politecnico di Torino 2018
4
5 — Multiply two sparse circulant binary matrices stored in a memory as the
6 — positions of set bits in their first row, and return a "tentative" result
7 — (which is unordered and not simplified modulo 2) in another location in
8 — memory.
9 — Controlling circuitry should take care of keeping all inputs in the
10 — "function arguments" section constant until the module reported back,
11 — and of ensuring validity of such inputs (no overlapping memory ranges and
12 — so on).
13 — The "start_i" signal should be kept high until "mult_done_o" goes high,
14 — implementing a rudimentary handshake, but this is not strictly required if
15 — the control circuitry operates on the same clock.
16
17 library IEEE;
18     use IEEE.std_logic_1164.all;
19     use IEEE.numeric_std.all;
20 library work;
21     use work.system_params.all;
22     use work.matrix_types.all;
23     use work.matrix_mult_functions.all;
24
25 entity circulant_multiplication is
26     port(
27         — control signals
28         clk_i:          in  std_logic;
29         rst_n_i:        in  std_logic;
30         start_i:        in  std_logic;
31         — function arguments in (not latched)
32         a_limit_i:      in  natural;    — number of elements in 1st matrix
33         a_base_i:       in  addr;       — base address of 1st matrix
34         b_limit_i:      in  natural;    — number of elements in 2nd matrix
35         b_base_i:       in  addr;       — base address of 2nd matrix
36         z_base_i:       in  addr;       — base address of result matrix
```

```

37      -- data, addresses and controls to memory
38      a_i:          in  pos;          -- operand from 1st matrix
39      a_addr_o:     out addr;         -- address for 1st matrix
40      b_i:          in  pos;          -- operand from 2nd matrix
41      b_addr_o:     out addr;         -- address for 2nd matrix
42      z_o:          out pos;          -- Ltr result
43      z_addr_o:     out addr;         -- address for Ltr result
44      wr_o:         out std_logic;    -- write enable for memory
45      -- report back once done
46      z_limit_o:    out natural;      -- number of elements in result matrix
47      mult_done_o:  out std_logic     -- high when done
48  );
49  end entity circulant_multiplication;
50
51  architecture rtl of circulant_multiplication is
52
53      -- assume inputs are kept constant by higher level state machine so we do
54      -- not need to sample them on start
55
56      signal      a_index:            natural;
57      signal      b_index:            natural;
58      signal      z_index:            natural;
59      signal      next_a_index:        natural;
60      signal      next_b_index:        natural;
61      signal      next_z_index:        natural;
62
63      type state_t is (IDLE, BUSY, DONE);
64      signal      state:               state_t;
65      signal      next_state:          state_t;
66
67  begin
68
69      z_o      <=  a_i + b_i when a_i + b_i < P else
70                (a_i + b_i) mod P;
71
72      a_addr_o <=  a_base_i + a_index;
73      b_addr_o <=  b_base_i + b_index;
74      z_addr_o <=  z_base_i + z_index;
75
76      -- count number of elements of result matrix, this can be done with a
77      -- multiplier but we are not in a hurry and do not want a big footprint.
78      -- Using z_index we get that for free!
79      z_limit_o <=  z_index;
80
81      state_comb: process(
82          state, start_i, a_limit_i, b_limit_i, next_a_index, next_b_index
83      )
84      begin
85          case state is
86              when IDLE =>
87                  if start_i = '1' then
88                      next_state <= BUSY;
89                  else
90                      next_state <= IDLE;
91                  end if;
92              when BUSY =>
93                  -- exit this state only once all combinations have been done
94                  if b_index /= b_limit_i or a_index /= a_limit_i then
95                      next_state <= BUSY;
96                  else
97                      next_state <= DONE;
98                  end if;

```

```

99         when DONE =>
100             if start_i = '0' then
101                 next_state <= IDLE;
102             else
103                 next_state <= DONE;
104             end if;
105         when others =>
106             next_state <= IDLE;
107     end case;
108 end process state_comb;
109
110 state_seq: process(rst_n_i, clk_i)
111 begin
112     if rst_n_i = '0' then
113         state <= IDLE;
114     elsif rising_edge(clk_i) then
115         state <= next_state;
116     end if;
117 end process state_seq;
118
119 output_comb: process(
120     state, a_index, b_index, z_index
121 )
122 begin
123     case state is
124     when IDLE =>
125         next_a_index <= 0;
126         next_b_index <= 0;
127         next_z_index <= 0;
128         wr_o <= '0';
129         mult_done_o <= '0';
130     when BUSY =>
131         if a_index < a_limit_i then
132             next_a_index <= a_index + 1;
133             next_b_index <= b_index;
134         else
135             next_a_index <= 0;
136             next_b_index <= b_index + 1;
137         end if;
138         next_z_index <= z_index + 1;
139         wr_o <= '1';
140         mult_done_o <= '0';
141     when DONE =>
142         next_a_index <= a_index;
143         next_b_index <= b_index;
144         next_z_index <= z_index;
145         wr_o <= '0';
146         mult_done_o <= '1';
147     when others => -- behave like IDLE
148         next_a_index <= 0;
149         next_b_index <= 0;
150         next_z_index <= 0;
151         wr_o <= '0';
152         mult_done_o <= '0';
153     end case;
154 end process output_comb;
155
156 output_seq: process(clk_i, rst_n_i)
157 begin
158     if rst_n_i = '0' then
159         a_index <= 0;
160         b_index <= 0;

```

```

161         z_index    <= 0;
162     elsif rising_edge(clk_i) then
163         a_index    <= next_a_index;
164         b_index    <= next_b_index;
165         z_index    <= next_z_index;
166     end if;
167 end process output_seq;
168
169 end architecture rtl;

```

## A.1.1 Sorting

```

1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4  library work;
5  use work.system_params.all;
6  use work.matrix_types.all;
7  use work.matrix_mult_functions.all;
8
9  entity sort is
10     port(
11         -- control logic
12         clk_i:          in  std_logic;
13         rst_n_i:        in  std_logic;
14         start_i:        in  std_logic;
15         -- function arguments in (not latched)
16         limit_i:        in  natural;    -- number of elements in array to sort
17         base_i:         in  addr;       -- base addr of array to sort
18         -- data, addresses and controls to memory
19         a_i:            in  pos;        -- element A from array
20         a_addr_o:       out addr;       -- address of element A
21         b_i:            in  pos;        -- element B from array
22         b_addr_o:       out addr;       -- address of element B
23         z_o:            out pos;        -- element Z to array
24         z_addr_o:       out addr;       -- address of element Z
25         wr_o:           out std_logic;  -- write enable for memory
26         -- report back once done
27         sort_done_o:    out std_logic   -- high when done
28     );
29 end entity sort;
30
31 architecture rtl of sort is
32
33     -- element A will be overwritten in the array, so we need to sample it in
34     -- order to insert it in the right place later on
35     signal tmp:         pos;
36     signal next_tmp:    pos;
37
38     -- iterators
39     signal i:           natural;
40     signal j:           natural;
41     signal next_i:      natural;
42     signal next_j:      natural;
43
44     -- state machine
45     type state_t is (
46         IDLE, PREP_INNER, CHECK_INNER, LOOP_INNER,
47         LAST_INNER, SAVE_0, SAVE_1, DONE

```

```

48 );
49 signal state: state_t;
50 signal next_state: state_t;
51
52 begin
53
54     b_addr_o <= base_i + to_unsigned(j, address_bits);
55
56     state_comb: process(state, start_i, next_j, next_i, a_i, b_i)
57     begin
58         case state is
59             when IDLE =>
60                 if start_i = '1' then
61                     next_state <= PREP_INNER;
62                 else
63                     next_state <= IDLE;
64                 end if;
65             when PREP_INNER =>
66                 next_state <= CHECK_INNER;
67             when CHECK_INNER =>
68                 if j /= 0 and b_i > tmp then
69                     next_state <= LOOP_INNER;
70                 elsif j = 0 and b_i > tmp then
71                     next_state <= LAST_INNER;
72                 else
73                     next_state <= SAVE_1;
74                 end if;
75             when LOOP_INNER =>
76                 if next_j /= 0 and a_i > tmp then
77                     next_state <= LOOP_INNER;
78                 elsif next_j = 0 and a_i > tmp then
79                     next_state <= LAST_INNER;
80                 else
81                     next_state <= SAVE_1;
82                 end if;
83             when LAST_INNER =>
84                 next_state <= SAVE_0;
85             when SAVE_0 =>
86                 if i /= limit_i - 1 then
87                     next_state <= PREP_INNER;
88                 else
89                     next_state <= DONE;
90                 end if;
91             when SAVE_1 =>
92                 if i /= limit_i - 1 then
93                     next_state <= PREP_INNER;
94                 else
95                     next_state <= DONE;
96                 end if;
97             when DONE =>
98                 if start_i = '0' then
99                     next_state <= IDLE;
100                 else
101                     next_state <= DONE;
102                 end if;
103             when others =>
104                 next_state <= IDLE;
105         end case;
106     end process state_comb;
107
108     state_seq: process(rst_n_i, clk_i)
109     begin

```

```

110         if rst_n_i = '0' then
111             state <= next_state;
112         elsif rising_edge(clk_i) then
113             state <= next_state;
114         end if;
115     end process state_seq;
116
117     output_comb: process(state, a_i, b_i, tmp, i, j)
118     begin
119         case state is
120             when IDLE =>
121                 -- internal signals
122                 next_tmp <= tmp;
123                 next_i <= 1;
124                 next_j <= 0;
125                 -- outputs
126                 a_addr_o <= base_i + to_unsigned(i, address_bits);
127                 z_addr_o <= base_i + to_unsigned(j + 1, address_bits);
128                 z_o <= b_i;
129                 wr_o <= '0';
130                 sort_done_o <= '0';
131             when PREP_INNER =>
132                 -- internal signals
133                 next_tmp <= a_i;
134                 next_i <= i;
135                 next_j <= i - 1;
136                 -- outputs
137                 a_addr_o <= base_i + to_unsigned(i, address_bits);
138                 z_addr_o <= base_i + to_unsigned(j + 1, address_bits);
139                 z_o <= b_i;
140                 wr_o <= '0';
141                 sort_done_o <= '0';
142             when CHECK_INNER =>
143                 -- internal signals
144                 next_tmp <= tmp;
145                 next_i <= i;
146                 next_j <= j;
147                 -- outputs
148                 a_addr_o <= base_i + to_unsigned(i, address_bits);
149                 z_addr_o <= base_i + to_unsigned(j + 1, address_bits);
150                 z_o <= b_i;
151                 wr_o <= '0';
152                 sort_done_o <= '0';
153             when LOOP_INNER =>
154                 -- internal signals
155                 next_tmp <= tmp;
156                 next_i <= i;
157                 next_j <= j - 1;
158                 -- outputs
159                 a_addr_o <= base_i + to_unsigned(j - 1, address_bits);
160                 z_addr_o <= base_i + to_unsigned(j + 1, address_bits);
161                 z_o <= b_i;
162                 wr_o <= '1';
163                 sort_done_o <= '0';
164             when LAST_INNER =>
165                 -- internal signals
166                 next_tmp <= tmp;
167                 next_i <= i;
168                 next_j <= j;
169                 -- outputs
170                 a_addr_o <= base_i + to_unsigned(i, address_bits);
171                 z_addr_o <= base_i + to_unsigned(j + 1, address_bits);

```



```

172         z_o      <= b_i;
173         wr_o      <= '1';
174         sort_done_o <= '0';
175     when SAVE_0 =>
176         -- internal signals
177         next_tmp   <= tmp;
178         next_i     <= i + 1;
179         next_j     <= j;
180         -- outputs
181         a_addr_o   <= base_i + to_unsigned(i, address_bits);
182         z_addr_o   <= base_i + to_unsigned(j, address_bits);
183         z_o        <= tmp;
184         wr_o       <= '1';
185         sort_done_o <= '0';
186     when SAVE_1 =>
187         -- internal signals
188         next_tmp   <= tmp;
189         next_i     <= i + 1;
190         next_j     <= j;
191         -- outputs
192         a_addr_o   <= base_i + to_unsigned(i, address_bits);
193         z_addr_o   <= base_i + to_unsigned(j + 1, address_bits);
194         z_o        <= tmp;
195         wr_o       <= '1';
196         sort_done_o <= '0';
197     when DONE =>
198         -- internal signals
199         next_tmp   <= tmp;
200         next_i     <= i;
201         next_j     <= j;
202         -- outputs
203         a_addr_o   <= base_i + to_unsigned(i, address_bits);
204         z_addr_o   <= base_i + to_unsigned(j + 1, address_bits);
205         z_o        <= tmp;
206         wr_o       <= '0';
207         sort_done_o <= '1';
208     when others => -- behave like IDLE
209         -- internal signals
210         next_tmp   <= tmp;
211         next_i     <= 1;
212         next_j     <= 0;
213         -- outputs
214         a_addr_o   <= base_i + to_unsigned(i, address_bits);
215         z_addr_o   <= base_i + to_unsigned(j + 1, address_bits);
216         z_o        <= b_i;
217         wr_o       <= '0';
218         sort_done_o <= '0';
219     end case;
220 end process output_comb;
221
222 output_seq: process(rst_n_i, clk_i)
223 begin
224     if rst_n_i = '0' then
225         tmp      <= (others => '0');
226         i        <= 0;
227         j        <= 0;
228     elsif rising_edge(clk_i) then
229         tmp      <= next_tmp;
230         i        <= next_i;
231         j        <= next_j;
232     end if;
233 end process output_seq;

```

```

234
235 end architecture rtl;

```

## A.1.2 Circulant sum

```

1  library IEEE;
2      use IEEE.std_logic_1164.all;
3      use IEEE.numeric_std.all;
4  library work;
5      use work.system_params.all;
6      use work.matrix_types.all;
7      use work.matrix_mult_functions.all;
8
9  entity circulant_sum is
10     port(
11         -- control signals
12         clk_i:      in  std_logic;
13         rst_n_i:    in  std_logic;
14         start_i:    in  std_logic;
15         -- function arguments in (not latched)
16         a_limit_i:  in  natural;    -- number of elements in 1st matrix
17         a_base_i:   in  addr;       -- base address of 1st matrix
18         b_limit_i:  in  natural;    -- number of elements in 2nd matrix
19         b_base_i:   in  addr;       -- base address of 2nd matrix
20         z_base_i:   in  addr;       -- base address of result matrix
21         -- data, addresses and controls to memory
22         a_i:        in  pos;
23         a_addr_o:    out addr;
24         b_i:        in  pos;
25         b_addr_o:    out addr;
26         z_o:        out pos;
27         z_addr_o:    out addr;
28         wr_o:       out std_logic; -- write enable for memory
29         -- report back once done
30         z_limit_o:   out natural;   -- number of remaining elements (minus one)
31         sum_done_o:  out std_logic
32     );
33 end entity circulant_sum;
34
35 architecture rtl of circulant_sum is
36
37     constant  INVALID_POS:      pos:=      to_unsigned(P, position_bits);
38
39     signal    i:                 natural;
40     signal    j:                 natural;
41     signal    k:                 natural;
42     signal    next_i:            natural;
43     signal    next_j:            natural;
44     signal    next_k:            natural;
45
46     signal    i_not_done:        std_logic;
47     signal    j_not_done:        std_logic;
48     signal    neither_done:      std_logic;
49
50     type state_t is (IDLE, COMPARE, SKIP, COPY_I, COPY_J, FILL_INVALID, DONE);
51     signal    state:             state_t;
52     signal    next_state:        state_t;
53
54 begin

```

```

55
56 i_not_done      <= '1' when i < a_limit_i and a_i /= INVALID.POS else
57                '0';
58 j_not_done      <= '1' when j < b_limit_i and b_i /= INVALID.POS else
59                '0';
60 neither_done    <= i_not_done and j_not_done;
61
62 a_addr_o        <= a_base_i + i;
63 b_addr_o        <= b_base_i + j;
64 z_addr_o        <= z_base_i + k;
65
66 z_limit_o       <= k;
67
68 state_comb: process(
69     state, start_i, i_not_done, j_not_done, neither_done,
70     k, a_i, b_i
71 )
72 begin
73     case state is
74     when IDLE =>
75         if start_i = '1' then
76             next_state <= COMPARE;
77         else
78             next_state <= IDLE;
79         end if;
80     when COMPARE =>
81         if neither_done = '1' and a_i = b_i then
82             next_state <= SKIP;
83         elsif neither_done = '1' and a_i < b_i then
84             next_state <= COPY_I;
85         elsif neither_done = '1' and a_i > b_i then
86             next_state <= COPY_J;
87         elsif i_not_done = '1' and not j_not_done = '1' then
88             next_state <= COPY_I;
89         elsif j_not_done = '1' and i_not_done = '0' then
90             next_state <= COPY_J;
91         elsif i_not_done = '0' and j_not_done = '0' -- cont.
92             and k < sum(M)*DV then
93             next_state <= FILL_INVALID;
94         else
95             next_state <= DONE;
96         end if;
97     when SKIP =>
98         next_state <= COMPARE;
99     when COPY_I =>
100         next_state <= COMPARE;
101     when COPY_J =>
102         next_state <= COMPARE;
103     when FILL_INVALID =>
104         next_state <= COMPARE;
105     when DONE =>
106         if start_i = '0' then
107             next_state <= IDLE;
108         else
109             next_state <= DONE;
110         end if;
111     when others =>
112         next_state <= IDLE;
113     end case;
114 end process state_comb;
115
116 state_seq: process(clk_i, rst_n_i)

```

```

117 begin
118     if rst_n_i = '0' then
119         state <= IDLE;
120     elsif rising_edge(clk_i) then
121         state <= next_state;
122     end if;
123 end process state_seq;
124
125 output_comb: process(state, i, j, k, a_i, b_i)
126 begin
127     case state is
128     when IDLE =>
129         -- internal signals
130         next_i <= 0;
131         next_j <= 0;
132         next_k <= 0;
133         -- outputs
134         z_o <= INVALID_POS;
135         wr_o <= '0';
136         sum_done_o <= '0';
137     when COMPARE =>
138         -- internal signals
139         next_i <= i;
140         next_j <= j;
141         next_k <= k;
142         -- outputs
143         z_o <= INVALID_POS;
144         wr_o <= '0';
145         sum_done_o <= '0';
146     when SKIP =>
147         -- internal signals
148         next_i <= i + 1;
149         next_j <= j + 1;
150         next_k <= k;
151         -- outputs
152         z_o <= INVALID_POS;
153         wr_o <= '0';
154         sum_done_o <= '0';
155     when COPY_I =>
156         -- internal signals
157         next_i <= i + 1;
158         next_j <= j;
159         next_k <= k + 1;
160         -- outputs
161         z_o <= a_i;
162         wr_o <= '1';
163         sum_done_o <= '0';
164     when COPY_J =>
165         -- internal signals
166         next_i <= i;
167         next_j <= j + 1;
168         next_k <= k + 1;
169         -- outputs
170         z_o <= b_i;
171         wr_o <= '1';
172         sum_done_o <= '0';
173     when FILL_INVALID =>
174         -- internal signals
175         next_i <= i;
176         next_j <= j;
177         next_k <= k + 1;
178         -- outputs

```

```

179         z_o      <= INVALID_POS;
180         wr_o      <= '1';
181         sum_done_o <= '0';
182     when DONE =>
183         -- internal signals
184         next_i     <= i;
185         next_j     <= j;
186         next_k     <= k + 1;
187         -- outputs
188         z_o      <= INVALID_POS;
189         wr_o      <= '0';
190         sum_done_o <= '1';
191     when others =>
192         -- internal signals
193         next_i     <= 0;
194         next_j     <= 0;
195         next_k     <= 0;
196         -- outputs
197         z_o      <= INVALID_POS;
198         wr_o      <= '0';
199         sum_done_o <= '0';
200     end case;
201 end process output_comb;
202
203 output_seq: process(clk_i, rst_n_i)
204 begin
205     if rst_n_i = '0' then
206         i      <= 0;
207         j      <= 0;
208         k      <= 0;
209     elsif rising_edge(clk_i) then
210         i      <= next_i;
211         j      <= next_j;
212         k      <= next_k;
213     end if;
214 end process output_seq;
215
216 end architecture rtl;

```

### A.1.3 Memory copy

```

1  library IEEE;
2      use IEEE.std_logic_1164.all;
3      use IEEE.numeric_std.all;
4  library work;
5      use work.matrix_types.all;
6
7  entity mem_copy is
8      port (
9          -- control signals
10         clk_i:      in  std_logic;
11         rst_n_i:    in  std_logic;
12         start_i:    in  std_logic;
13         -- function arguments in (not latched)
14         limit_i:    in  natural;    -- number of elements in source array
15         a_base_i:    in  addr;      -- base addr of source array
16         z_base_i:    in  addr;      -- base addr of destination array
17         -- data, addresses and controls to memory
18         a_i:        in  pos;        -- element A from source array

```

```

19         a_addr_o:      out addr;      -- address of element A
20         z_o:           out pos;       -- element Z to destination array
21         z_addr_o:      out addr;      -- address of element Z
22         wr_o:           out std_logic; -- write enable for memory
23         -- report back once done
24         copy_done_o:    out std_logic  -- high when done
25     );
26 end entity mem_copy;
27
28 architecture rtl of mem_copy is
29
30     signal i:           natural;
31     signal next_i:      natural;
32
33     type state_t is (IDLE, COPY, DONE);
34     signal state:       state_t;
35     signal next_state:  state_t;
36
37 begin
38
39     a_addr_o <= a_base_i + i;
40     z_addr_o <= z_base_i + i;
41     z_o      <= a_i;
42
43     state_comb: process(state, start_i, i, limit_i)
44     begin
45         case state is
46             when IDLE =>
47                 if start_i = '1' then
48                     next_state <= COPY;
49                 else
50                     next_state <= IDLE;
51                 end if;
52             when COPY =>
53                 if i = limit_i-1 then
54                     next_state <= DONE;
55                 else
56                     next_state <= COPY;
57                 end if;
58             when DONE =>
59                 if start_i = '0' then
60                     next_state <= IDLE;
61                 else
62                     next_state <= DONE;
63                 end if;
64             when others =>
65                 next_state <= IDLE;
66         end case;
67     end process state_comb;
68
69     state_seq: process(clk_i, rst_n_i)
70     begin
71         if rst_n_i = '0' then
72             state <= IDLE;
73         elsif rising_edge(clk_i) then
74             state <= next_state;
75         end if;
76     end process state_seq;
77
78     output_comb: process(state, i)
79     begin
80         case state is

```

```

81         when IDLE =>
82             -- internal signals
83             next_i    <= 0;
84             -- outputs
85             wr_o       <= '0';
86             copy_done_o <= '0';
87         when COPY =>
88             -- internal signals
89             next_i    <= i + 1;
90             -- outputs
91             wr_o       <= '1';
92             copy_done_o <= '0';
93         when DONE =>
94             -- internal signals
95             next_i    <= 0;
96             -- outputs
97             wr_o       <= '0';
98             copy_done_o <= '1';
99         when others =>
100            -- internal signals
101            next_i    <= i;
102            -- outputs
103            wr_o       <= '0';
104            copy_done_o <= '0';
105         end case;
106     end process output_comb;
107
108     output_seq: process (clk_i, rst_n_i)
109     begin
110         if rst_n_i = '0' then
111             i <= 0;
112         elsif rising_edge(clk_i) then
113             i <= next_i;
114         end if;
115     end process output_seq;
116
117 end architecture rtl;

```

## A.2 Vector by matrix

### A.2.1 Vector by circulant

```

1  library IEEE;
2      use IEEE.std_logic_1164.all;
3      use IEEE.numeric_std.all;
4  library work;
5      use work.system_params.all;
6      use work.matrix_types.all;
7
8  entity vector_by_circulant is
9      port(
10         -- control signals
11         clk_i:          in  std_logic;
12         rst_n_i:        in  std_logic;
13         start_i:         in  std_logic;
14         binary_i:        in  std_logic;
15         -- function arguments in (not latched)

```

```

16         a_base_i:      in   addr;
17         b_base_i:      in   addr;
18         b_weight:      in   natural;
19         z_base_i:      in   addr;
20         op:             in   std_logic;  -- '0' -> z=aB; '1' -> z=Ba
21         -- data, addresses and controls to memory
22         a_i:           in   pos;
23         a_addr_o:      out  addr;
24         b_i:           in   pos;
25         b_addr_o:      out  addr;
26         z_o:           out  pos;
27         z_addr_o:      out  addr;
28         wr_o:          out  std_logic;
29         done_o:        out  std_logic
30     );
31 end entity vector_by_circulant;
32
33 architecture arch of vector_by_circulant is
34
35     type state_t is (IDLE, GET_M, INNER_LOOP, DONE);
36     signal state:    state_t;
37     signal next_state: state_t;
38
39     signal m_index_c, m_index_r:    natural;
40     signal m_c, m_r:    pos;
41     signal n_c, n_r:    addr;
42     signal i_c, i_r:    addr;
43
44 begin
45
46     comb: process(state, start_i, binary_i,
47         m_index_r, m_c, m_r, n_c, n_r, i_c, i_r,
48         op, b_weight, a_base_i, b_base_i, z_base_i,
49         a_i, b_i)
50     begin
51         next_state <= state;
52         wr_o <= '0';
53         done_o <= '0';
54         m_index_c <= m_index_r;
55         a_addr_o <= to_unsigned(0, address_bits);
56         b_addr_o <= to_unsigned(0, address_bits);
57         z_o <= to_unsigned(0, position_bits);
58         z_addr_o <= to_unsigned(0, address_bits);
59         m_c <= m_r;
60         n_c <= n_r;
61         i_c <= i_r;
62         case state is
63             when IDLE =>
64                 m_index_c <= 0;
65                 m_c <= to_unsigned(0, position_bits);
66                 n_c <= to_unsigned(0, address_bits);
67                 i_c <= to_unsigned(0, address_bits);
68                 if start_i = '1' then
69                     next_state <= GET_M;
70                 end if;
71             when GET_M =>
72                 m_index_c <= m_index_r + 1;
73                 b_addr_o <= b_base_i + m_index_r;
74                 m_c <= b_i;
75                 n_c <= to_unsigned(0, address_bits);
76                 if op = '0' then
77                     i_c <= to_unsigned(to_integer(m_c), address_bits);

```



```

78         else
79             i_c      <=  to_unsigned(to_integer(P-m_c-1),  -- cont.
80                                     address_bits);
81         end if;
82         if b_i = P then
83             next_state <= DONE;
84         else
85             next_state <= INNER_LOOP;
86         end if;
87     when INNER_LOOP =>
88         wr_o      <=  '1';
89         n_c      <=  n_r + 1;
90         if i_r = P - 1 then
91             i_c <=  to_unsigned(0, address_bits);
92         else
93             i_c <=  i_r + 1;
94         end if;
95         a_addr_o  <=  a_base_i + n_c;
96         b_addr_o  <=  z_base_i + i_c;
97         z_addr_o  <=  z_base_i + i_c;
98         if binary_i = '0' then
99             z_o      <=  a_i + b_i;
100        else
101            z_o      <=  (0 => a_i(0) xor b_i(0), others => '0');
102        end if;
103        if n_r = P-1 then
104            if m_index_r = b_weight then
105                next_state <= DONE;
106            else
107                next_state <= GETM;
108            end if;
109        end if;
110    when DONE =>
111        done_o <=  '1';
112        if start_i = '0' then
113            next_state <= IDLE;
114        end if;
115    when others =>
116        next_state <= IDLE;
117    end case;
118 end process comb;
119
120 seq: process(rst_n_i, clk_i)
121 begin
122     if rst_n_i = '0' then
123         state      <=  IDLE;
124         m_index_r  <=  0;
125         m_r        <=  to_unsigned(0, position_bits);
126         n_r        <=  to_unsigned(0, address_bits);
127         i_r        <=  to_unsigned(0, address_bits);
128     elsif rising_edge(clk_i) then
129         state      <=  next_state;
130         m_index_r  <=  m_index_c;
131         m_r        <=  m_c;
132         n_r        <=  n_c;
133         i_r        <=  i_c;
134     end if;
135 end process seq;
136
137 end architecture;

```

## A.2.2 $\underline{x}$ by $L^T$

```

1  library IEEE;
2      use IEEE.std_logic_1164.all;
3      use IEEE.numeric_std.all;
4  library work;
5      use work.system_params.all;
6      use work.matrix_types.all;
7      use work.matrix_mult_functions.all;
8      use work.memory_map.all;
9
10 entity x_by_Ltr is
11     port (
12         -- control signals
13         clk_i:          in  std_logic;
14         rst_n_i:        in  std_logic;
15         start_i:        in  std_logic;
16         -- controls to multiplication core
17         vector_base_o:  out addr;
18         block_base_o:   out addr;
19         result_base_o:  out addr;
20         start_mul_o:    out std_logic;
21         mul_done_i:     in  std_logic;
22         -- report back
23         done_o:         out std_logic
24     );
25 end entity x_by_Ltr;
26
27 architecture rtl of x_by_Ltr is
28
29     type state_t is (IDLE, BUSY, DONE);
30     signal state: state_t;
31     signal next_state: state_t;
32
33     signal block_count_c, block_count_r: natural;
34
35     signal vector_base_c, vector_base_r: addr;
36     signal block_base_c, block_base_r: addr;
37
38 begin
39
40     vector_base_o <= vector_base_r;
41     block_base_o <= block_base_r;
42     result_base_o <= S_BASE_ADDR;
43
44     comb: process(state, start_i,
45         block_count_r, block_base_r, vector_base_r,
46         mul_done_i)
47     begin
48         start_mul_o <= '0';
49         next_state <= state;
50         block_count_c <= block_count_r;
51         block_base_c <= block_base_r;
52         vector_base_c <= vector_base_r;
53         done_o <= '0';
54         case state is
55             when IDLE =>
56                 block_count_c <= 0;
57                 block_base_c <= LTR_BASE_ADDR;
58                 vector_base_c <= X_BASE_ADDR;
59                 if start_i = '1' then

```

```

60         next_state <= BUSY;
61     end if;
62 when BUSY =>
63     start_mul_o <= '1';
64     if mul_done_i = '1' then
65         block_count_c <= block_count_r + 1;
66         -- M*Dv is the maximum number of elements in a block of Ltr
67         block_base_c <= block_base_r + sum(M)*DV;
68         vector_base_c <= vector_base_r + P;
69         start_mul_o <= '0';
70         if block_count_r = N0-1 then
71             next_state <= DONE;
72         end if;
73     end if;
74 when DONE =>
75     done_o <= '1';
76     if start_i = '0' then
77         next_state <= IDLE;
78     end if;
79 when others =>
80     next_state <= IDLE;
81 end case;
82 end process comb;
83
84 seq: process(clk_i, rst_n_i)
85 begin
86     if rst_n_i = '0' then
87         state <= IDLE;
88         block_count_r <= 0;
89         block_base_r <= to_unsigned(0, address_bits);
90         vector_base_r <= to_unsigned(0, address_bits);
91     elsif rising_edge(clk_i) then
92         state <= next_state;
93         block_count_r <= block_count_c;
94         block_base_r <= block_base_c;
95         vector_base_r <= vector_base_c;
96     end if;
97 end process seq;
98
99 end architecture rtl;

```

### A.2.3 $H^T$ by $\underline{s}^T$

```

1  library IEEE;
2      use IEEE.std_logic_1164.all;
3      use IEEE.numeric_std.all;
4  library work;
5      use work.system_params.all;
6      use work.matrix_types.all;
7      use work.matrix_mult_functions.all;
8      use work.memory_map.all;
9
10 entity Htr_by_str is
11     port (
12         -- control signals
13         clk_i:          in  std_logic;
14         rst_n_i:         in  std_logic;
15         start_i:         in  std_logic;
16         -- controls to multiplication core

```

```

17         vector_base_o: out addr;
18         block_base_o:  out addr;
19         result_base_o: out addr;
20         start_mul_o:   out std_logic;
21         mul_done_i:    in  std_logic;
22         -- report back
23         done_o:        out std_logic
24     );
25 end entity Htr_by_str;
26
27 architecture rtl of Htr_by_str is
28
29     type state_t is (IDLE, BUSY, DONE);
30     signal state: state_t;
31     signal next_state: state_t;
32
33     signal block_count_c, block_count_r: natural;
34
35     signal vector_base_c, vector_base_r: addr;
36     signal block_base_c, block_base_r:  addr;
37     signal result_base_c, result_base_r: addr;
38
39 begin
40
41     vector_base_o <= vector_base_r;
42     block_base_o  <= block_base_r;
43     result_base_o <= result_base_r;
44
45     comb: process(state, start_i,
46         block_count_r, block_base_r, vector_base_r, result_base_r,
47         mul_done_i)
48     begin
49         start_mul_o <= '0';
50         next_state <= state;
51         block_count_c <= block_count_r;
52         block_base_c <= block_base_r;
53         vector_base_c <= vector_base_r;
54         result_base_c <= result_base_r;
55         done_o <= '0';
56         case state is
57             when IDLE =>
58                 block_count_c <= 0;
59                 block_base_c <= HTR_BASE_ADDR;
60                 vector_base_c <= S_BASE_ADDR;
61                 result_base_c <= SIGMA_BASE_ADDR;
62                 if start_i = '1' then
63                     next_state <= BUSY;
64                 end if;
65             when BUSY =>
66                 start_mul_o <= '1';
67                 if mul_done_i = '1' then
68                     block_count_c <= block_count_r + 1;
69                     block_base_c <= block_base_r + DV;
70                     vector_base_c <= vector_base_r + P;
71                     result_base_c <= result_base_r + P;
72                     start_mul_o <= '0';
73                     if block_count_r = N0-1 then
74                         next_state <= DONE;
75                     end if;
76                 end if;
77             when DONE =>
78                 done_o <= '1';

```

```

79         if start_i = '0' then
80             next_state <= IDLE;
81         end if;
82         when others =>
83             next_state <= IDLE;
84     end case;
85 end process comb;
86
87 seq: process(clk_i, rst_n_i)
88 begin
89     if rst_n_i = '0' then
90         state <= IDLE;
91         block_count_r <= 0;
92         block_base_r <= HTR_BASE_ADDR;
93         vector_base_r <= S_BASE_ADDR;
94         result_base_r <= SIGMA_BASE_ADDR;
95     elsif rising_edge(clk_i) then
96         state <= next_state;
97         block_count_r <= block_count_c;
98         block_base_r <= block_base_c;
99         vector_base_r <= vector_base_c;
100        result_base_r <= result_base_c;
101    end if;
102 end process seq;
103
104 end architecture rtl;

```

#### A.2.4 $Q^T$ by $\underline{\Sigma}^T$

```

1  library IEEE;
2      use IEEE.std_logic_1164.all;
3      use IEEE.numeric_std.all;
4  library work;
5      use work.system_params.all;
6      use work.matrix_types.all;
7      use work.matrix_mult_functions.all;
8      use work.memory_map.all;
9
10 entity Qtr_by_sigmatr is
11     port (
12         -- control signals
13         clk_i: in std_logic;
14         rst_n_i: in std_logic;
15         start_i: in std_logic;
16         -- controls to multiplication core
17         vector_base_o: out addr;
18         block_base_o: out addr;
19         block_weight_o: out natural;
20         result_base_o: out addr;
21         start_mul_o: out std_logic;
22         mul_done_i: in std_logic;
23         -- report back
24         done_o: out std_logic
25     );
26 end entity Qtr_by_sigmatr;
27
28 architecture rtl of Qtr_by_sigmatr is
29
30     type state_t is (IDLE, BUSY, DONE);

```

```

31  signal state: state_t;
32  signal next_state: state_t;
33
34  signal block_row_c, block_row_r: natural;
35  signal block_col_c, block_col_r: natural;
36
37  signal Qtr_block_base_map:
38      addr_matrix(N0-1 downto 0, N0-1 downto 0) := get_Qtr_block_base;
39  signal Qtr_block_limit_map:
40      n_matrix(N0-1 downto 0, N0-1 downto 0) := get_block_limits;
41
42  signal vector_base_c, vector_base_r: addr;
43  signal block_base_c, block_base_r: addr;
44  signal result_base_c, result_base_r: addr;
45
46  begin
47
48      vector_base_o <= vector_base_r;
49      block_base_o <= block_base_r;
50      block_weight_o <= Qtr_block_limit_map(block_row_r, block_col_r);
51      result_base_o <= result_base_r;
52
53      comb: process(state, start_i,
54          block_base_r, vector_base_r, result_base_r,
55          block_row_r, block_col_r,
56          mul_done_i)
57      begin
58          start_mul_o <= '0';
59          next_state <= state;
60          block_row_c <= block_row_r;
61          block_col_c <= block_col_r;
62          block_base_c <= block_base_r;
63          vector_base_c <= vector_base_r;
64          result_base_c <= result_base_r;
65          done_o <= '0';
66          case state is
67              when IDLE =>
68                  block_row_c <= 0;
69                  block_col_c <= 0;
70                  block_base_c <= QTR_BASE_ADDR;
71                  vector_base_c <= SIGMA_BASE_ADDR;
72                  result_base_c <= R_BASE_ADDR;
73                  if start_i = '1' then
74                      next_state <= BUSY;
75                  end if;
76              when BUSY =>
77                  start_mul_o <= '1';
78                  if mul_done_i = '1' then
79                      start_mul_o <= '0';
80                      if block_col_r /= N0-1 then
81                          block_col_c <= block_col_r + 1;
82                          block_base_c <= QTR_BASE_ADDR + — cont.
83                              Qtr_block_base_map(block_row_r, block_col_r);
84                          vector_base_c <= vector_base_r + P;
85                          result_base_c <= result_base_r;
86                      else
87                          if block_row_r /= N0-1 then
88                              block_row_c <= block_row_r + 1;
89                              block_col_c <= 0;
90                              block_base_c <= QTR_BASE_ADDR + — cont.
91                                  Qtr_block_base_map(block_row_r + 1, — cont.
92                                      block_col_r);

```

```

93         vector_base_c <= SIGMA_BASE_ADDR;
94         result_base_c <= result_base_r + P;
95     else
96         next_state <= DONE;
97     end if;
98 end if;
99 end if;
100 when DONE =>
101     done_o <= '1';
102     if start_i = '0' then
103         next_state <= IDLE;
104     end if;
105 when others =>
106     next_state <= IDLE;
107 end case;
108 end process comb;
109
110 seq: process(clk_i, rst_n_i)
111 begin
112     if rst_n_i = '0' then
113         state <= IDLE;
114         block_row_r <= 0;
115         block_col_r <= 0;
116         block_base_r <= HTR_BASE_ADDR;
117         vector_base_r <= S_BASE_ADDR;
118         result_base_r <= SIGMA_BASE_ADDR;
119     elsif rising_edge(clk_i) then
120         state <= next_state;
121         block_base_r <= block_base_c;
122         vector_base_r <= vector_base_c;
123         result_base_r <= result_base_c;
124         block_row_r <= block_row_c;
125         block_col_r <= block_col_c;
126     end if;
127 end process seq;
128
129 end architecture rtl;

```

## A.2.5 $\underline{e}$ by $H^T$

```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.numeric_std.all;
4 library work;
5 use work.system_params.all;
6 use work.matrix_types.all;
7 use work.matrix_mult_functions.all;
8 use work.memory_map.all;
9
10 entity e_by_Htr is
11 port (
12     -- control signals
13     clk_i: in std_logic;
14     rst_n_i: in std_logic;
15     start_i: in std_logic;
16     -- controls to multiplication core
17     vector_base_o: out addr;
18     block_base_o: out addr;
19     result_base_o: out addr;

```

```

20         start_mul_o:    out std_logic;
21         mul_done_i:     in  std_logic;
22         -- report back
23         done_o:         out std_logic
24     );
25 end entity e_by_Htr;
26
27 architecture rtl of e_by_Htr is
28
29     type state_t is (IDLE, BUSY, DONE);
30     signal state:   state_t;
31     signal next_state: state_t;
32
33     signal block_count_c, block_count_r:   natural;
34
35     signal vector_base_c, vector_base_r:   addr;
36     signal block_base_c, block_base_r:     addr;
37
38 begin
39
40     vector_base_o <= vector_base_r;
41     block_base_o  <= block_base_r;
42     result_base_o <= S.BASE_ADDR;
43
44     comb: process(state, start_i,
45         block_count_r, block_base_r, vector_base_r,
46         mul_done_i)
47     begin
48         start_mul_o <= '0';
49         next_state <= state;
50         block_count_c <= block_count_r;
51         block_base_c <= block_base_r;
52         vector_base_c <= vector_base_r;
53         done_o <= '0';
54         case state is
55             when IDLE =>
56                 block_count_c <= 0;
57                 block_base_c <= HTR.BASE_ADDR;
58                 vector_base_c <= E.BASE_ADDR;
59                 if start_i = '1' then
60                     next_state <= BUSY;
61                 end if;
62             when BUSY =>
63                 start_mul_o <= '1';
64                 if mul_done_i = '1' then
65                     block_count_c <= block_count_r + 1;
66                     block_base_c <= block_base_r + DV;
67                     vector_base_c <= vector_base_r + P;
68                     start_mul_o <= '0';
69                     if block_count_r = N0-1 then
70                         next_state <= DONE;
71                     end if;
72                 end if;
73             when DONE =>
74                 done_o <= '1';
75                 if start_i = '0' then
76                     next_state <= IDLE;
77                 end if;
78             when others =>
79                 next_state <= IDLE;
80         end case;
81     end process comb;

```



```

82
83     seq: process(clk_i, rst_n_i)
84     begin
85         if rst_n_i = '0' then
86             state <= IDLE;
87             block_count_r <= 0;
88             block_base_r <= to_unsigned(0, address_bits);
89             vector_base_r <= to_unsigned(0, address_bits);
90         elsif rising_edge(clk_i) then
91             state <= next_state;
92             block_count_r <= block_count_c;
93             block_base_r <= block_base_c;
94             vector_base_r <= vector_base_c;
95         end if;
96     end process seq;
97
98 end architecture rtl;

```

## A.3 Error update

### Peak search

```

1  library IEEE;
2      use IEEE.std_logic_1164.all;
3      use IEEE.numeric_std.all;
4  library work;
5      use work.system_params.all;
6      use work.matrix_types.all;
7      use work.matrix_mult_functions.all;
8      use work.memory_map.all;
9
10 entity find_max is
11     port(
12         -- control signals
13         clk_i:      in  std_logic;
14         rst_n_i:    in  std_logic;
15         start_i:    in  std_logic;
16         -- to memory
17         a_i:        in  pos;
18         a_addr_o:    out addr;
19         b_i:        in  pos;
20         b_addr_o:    out addr;
21         z_o:        out pos;
22         z_addr_o:    out addr;
23         wr_o:       out std_logic;
24         -- report back when done
25         max_idx_o:   out n_array(15 downto 0);
26         done_o:      out std_logic
27     );
28 end entity find_max;
29
30 architecture rtl of find_max is
31
32     type state_t is (IDLE, BUSY, DONE);
33     signal state: state_t;
34     signal next_state: state_t;
35

```

```

36     signal a_index_c, a_index_r:    natural;
37
38     signal max_val_c, max_val_r:      pos;
39     signal max_indexes_c, max_indexes_r:  n_array(15 downto 0);
40
41     signal i_c, i_r:    natural;
42
43     constant INVALID:    natural:=  N0*P;
44
45 begin
46
47     a_addr_o    <=  RBASE_ADDR + a_index_r;
48     b_addr_o    <=  to_unsigned(0, address_bits);
49     z_o         <=  to_unsigned(0, position_bits);
50     z_addr_o    <=  to_unsigned(0, address_bits);
51     wr_o        <=  '0';
52     max_idx_o   <=  max_indexes_r;
53
54     seq: process(clk_i, rst_n_i)
55     begin
56         if rst_n_i = '0' then
57             state <= IDLE;
58             a_index_r <= 0;
59             max_val_r <= to_unsigned(0, position_bits);
60             max_indexes_r <= (others => invalid);
61             i_r <= 0;
62         elsif rising_edge(clk_i) then
63             state <= next_state;
64             a_index_r <= a_index_c;
65             max_val_r <= max_val_c;
66             max_indexes_r <= max_indexes_c;
67             i_r <= i_c;
68         end if;
69     end process seq;
70     comb: process(
71         state, start_i,
72         a_index_r, a_i,
73         max_val_r, max_indexes_r, i_r)
74     begin
75         next_state <= state;
76         done_o <= '0';
77         a_index_c <= 0;
78         max_val_c <= max_val_r;
79         max_indexes_c <= max_indexes_r;
80         i_c <= i_r;
81         case state is
82             when IDLE =>
83                 max_val_c <= to_unsigned(0, position_bits);
84                 max_indexes_c <= (others => INVALID);
85                 i_c <= 0;
86                 if start_i = '1' then
87                     next_state <= BUSY;
88                 end if;
89             when BUSY =>
90                 a_index_c <= a_index_r + 1;
91                 if a_i = max_val_r then
92                     max_indexes_c(i_r) <= a_index_r;
93                     -- if we have too many equal maxes we'll flip just some
94                     if i_r /= 15 then
95                         i_c <= i_r + 1;
96                     end if;
97                 elsif a_i > max_val_r then

```

```

98         max_val_c      <=  a_i;
99         max_indexes_c   <=  (0 => a_index_r, others => INVALID);
100         i_c             <=  1;
101     end if;
102     if a_index_r = N0*P-1 then
103         a_index_c      <=  0;
104         next_state     <=  DONE;
105     end if;
106     when DONE =>
107         done_o <= '1';
108         if start_i = '0' then
109             next_state <= IDLE;
110         end if;
111     end case;
112 end process comb;
113
114 end architecture rtl;

```

## Vector plus rows

```

1  library IEEE;
2      use IEEE.std_logic_1164.all;
3      use IEEE.numeric_std.all;
4  library work;
5      use work.system_params.all;
6      use work.matrix_types.all;
7      use work.memory_map.all;
8      use work.matrix_mult_functions.all;
9
10 entity vector_plus_rows is
11     port (
12         -- control signals
13         clk_i:      in  std_logic;
14         rst_n_i:    in  std_logic;
15         start_i:    in  std_logic;
16         -- row indexes
17         row_idx_i:  in  n_array(15 downto 0);
18         -- to memory
19         a_i:        in  pos;
20         a_addr_o:   out addr;
21         b_i:        in  pos;
22         b_addr_o:   out addr;
23         z_o:        out pos;
24         z_addr_o:   out addr;
25         wr_o:       out std_logic;
26         -- report back when done
27         done_o:     out std_logic
28     );
29 end entity vector_plus_rows;
30
31 architecture rtl of vector_plus_rows is
32
33     type state_t is (IDLE, RELATIVE_ROW, SUMROW, DONE);
34     signal state: state_t;
35     signal next_state: state_t;
36
37     signal cur_idx_c: natural;
38     signal cur_idx_r: natural;
39

```

```

40  signal block_row_c:    natural;
41  signal block_row_r:    natural;
42  signal block_col_c:    natural;
43  signal block_col_r:    natural;
44  signal rel_row_c:      natural;
45  signal rel_row_r:      natural;
46
47  signal i_c:            natural;
48  signal i_r:            natural;
49
50  signal Qtr_block_base:  addr;
51  signal Qtr_block_count: natural;
52
53  constant Qtr_block_limit_map:  n_matrix:=  get_block_limits;
54  constant Qtr_block_base_map:   addr_matrix:= get_Qtr_block_base;
55
56  constant INVALID:    natural:=  N0*P;
57
58  begin
59
60  Qtr_block_count <= Qtr_block_limit_map(block_row_r, block_col_r);
61  Qtr_block_base  <= Qtr_block_base_map(block_row_r, block_col_r);
62
63  a_addr_o <= EBASE_ADDR + block_col_r*P + ((b_i + rel_row_r) mod P);
64  b_addr_o <= QTR_BASE_ADDR + Qtr_block_base + i_r;
65  z_o <= (a_i + 1) mod 2;
66  z_addr_o <= EBASE_ADDR + block_col_r*P + ((b_i + rel_row_r) mod P);
67
68  seq: process(clk_i, rst_n_i)
69  begin
70      if rst_n_i = '0' then
71          state <= IDLE;
72          cur_idx_r <= 0;
73          block_row_r <= 0;
74          block_col_r <= 0;
75          rel_row_r <= 0;
76          i_r <= 0;
77      elsif rising_edge(clk_i) then
78          state <= next_state;
79          cur_idx_r <= cur_idx_c;
80          block_row_r <= block_row_c;
81          block_col_r <= block_col_c;
82          rel_row_r <= rel_row_c;
83          i_r <= i_c;
84      end if;
85  end process seq;
86  comb: process(
87      state, start_i,
88      block_row_r, block_col_r, rel_row_r,
89      row_idx_i, cur_idx_r, i_r,
90      Qtr_block_count)
91  begin
92      next_state <= state;
93      cur_idx_c <= cur_idx_r;
94      block_row_c <= block_row_r;
95      block_col_c <= block_col_r;
96      rel_row_c <= rel_row_r;
97      i_c <= 0;
98      wr_o <= '0';
99      done_o <= '0';
100  case (state) is
101      when IDLE =>

```

```

102         cur_idx_c <= 0;
103         block_row_c <= 0;
104         block_col_c <= 0;
105         if start_i = '1' then
106             next_state <= RELATIVEROW;
107         end if;
108     when RELATIVEROW =>
109         if row_idx_i(cur_idx_r) /= INVALID then
110             block_row_c <= row_idx_i(cur_idx_r) / P;
111             rel_row_c <= row_idx_i(cur_idx_r) mod P;
112             next_state <= SUMROW;
113         else
114             next_state <= DONE;
115         end if;
116     when SUMROW =>
117         wr_o <= '1';
118         if i_r /= Qtr_block_count then
119             i_c <= i_r + 1;
120         else
121             if block_col_r /= N0-1 then
122                 block_col_c <= block_col_r + 1;
123             else
124                 if cur_idx_r = 15 then
125                     next_state <= DONE;
126                 else
127                     cur_idx_c <= cur_idx_r + 1;
128                     block_col_c <= 0;
129                     next_state <= RELATIVEROW;
130                 end if;
131             end if;
132         end if;
133     when DONE =>
134         done_o <= '1';
135         if start_i = '0' then
136             next_state <= IDLE;
137         end if;
138     end case;
139 end process comb;
140
141 end architecture rtl;

```

## A.4 Loop control and message computation

### Zero syndrome detection

```

1  library IEEE;
2      use IEEE.std_logic_1164.all;
3      use IEEE.numeric_std.all;
4  library work;
5      use work.system_params.all;
6      use work.matrix_types.all;
7      use work.matrix_mult_functions.all;
8      use work.memory_map.all;
9
10 entity null_syndrome is
11     port(
12         -- control signals

```

```

13         clk_i:      in  std_logic;
14         rst_n_i:    in  std_logic;
15         start_i:    in  std_logic;
16         -- to memory
17         a_i:        in  pos;
18         a_addr_o:    out addr;
19         b_i:        in  pos;
20         b_addr_o:    out addr;
21         z_o:        out pos;
22         z_addr_o:    out addr;
23         wr_o:       out std_logic;
24         -- report back when done
25         null_syn_o:  out std_logic;
26         done_o:     out std_logic
27     );
28 end entity null_syndrome;
29
30 architecture rtl of null_syndrome is
31
32     type state_t is (IDLE, BUSY, DONE);
33     signal state:   state_t;
34     signal next_state: state_t;
35
36     signal bad_syn_c: std_logic;
37     signal bad_syn_r: std_logic;
38
39     signal i_c:      natural;
40     signal i_r:      natural;
41
42 begin
43
44     a_addr_o <= SBASE_ADDR + i_r;
45     b_addr_o <= SBASE_ADDR + i_r + 1;
46     z_o      <= to_unsigned(0, position_bits);
47     z_addr_o <= SBASE_ADDR;
48     wr_o     <= '0';
49     null_syn_o <= not bad_syn_r;
50
51     seq: process(clk_i, rst_n_i)
52     begin
53         if rst_n_i = '0' then
54             state <= IDLE;
55             bad_syn_r <= '0';
56             i_r <= 0;
57         elsif rising_edge(clk_i) then
58             state <= next_state;
59             bad_syn_r <= bad_syn_c;
60             i_r <= i_c;
61         end if;
62     end process seq;
63     comb: process(state, start_i, a_i, b_i, i_r, bad_syn_c, bad_syn_r)
64     begin
65         next_state <= state;
66         bad_syn_c <= bad_syn_r;
67         i_c <= i_r;
68         done_o <= '0';
69         case state is
70             when IDLE =>
71                 bad_syn_c <= '0';
72                 i_c <= 0;
73                 if start_i = '1' then
74                     next_state <= BUSY;

```

```

75         end if;
76     when BUSY =>
77         i_c    <= i_r + 2;
78         if a_i /= 0 then
79             bad_syn_c    <= '1';
80         end if;
81         if b_i /= 0 and i_r /= P-1 then
82             bad_syn_c    <= '1';
83         end if;
84         if bad_syn_c = '1' or i_r = P-1 then
85             next_state <= DONE;
86         end if;
87     when DONE =>
88         done_o    <= '1';
89         if start_i = '0' then
90             next_state <= IDLE;
91         end if;
92     end case;
93 end process comb;
94
95 end architecture rtl;

```

## Intermediate result clear

```

1  library IEEE;
2      use IEEE.std_logic_1164.all;
3      use IEEE.numeric_std.all;
4  library work;
5      use work.system_params.all;
6      use work.matrix_types.all;
7      use work.matrix_mult_functions.all;
8      use work.memory_map.all;
9
10 entity clear_temp is
11     port(
12         -- control signals
13         clk_i:    in  std_logic;
14         rst_n_i:  in  std_logic;
15         start_i:  in  std_logic;
16         -- to memory
17         a_i:      in  pos;
18         a_addr_o: out  addr;
19         b_i:      in  pos;
20         b_addr_o: out  addr;
21         z_o:      out  pos;
22         z_addr_o: out  addr;
23         wr_o:     out  std_logic;
24         -- report back when done
25         done_o:   out  std_logic
26     );
27 end entity clear_temp;
28
29 architecture rtl of clear_temp is
30
31     type state_t is (
32         IDLE, CLEAR_MULRES, CLEAR_SUM_TMP, CLEAR_SIGMA,
33         CLEAR_R, DONE
34     );
35     signal state: state_t;

```

```

36     signal next_state: state_t;
37
38     signal i_c: integer;
39     signal i_r: integer;
40
41 begin
42
43     a_addr_o <= HTR_BASE_ADDR;
44     b_addr_o <= HTR_BASE_ADDR;
45     z_o <= to_unsigned(0, position_bits);
46
47     seq: process (clk_i, rst_n_i)
48     begin
49         if rst_n_i = '0' then
50             state <= IDLE;
51             i_r <= 0;
52         elsif rising_edge(clk_i) then
53             state <= next_state;
54             i_r <= i_c;
55         end if;
56     end process seq;
57     comb: process (state, start_i, i_r)
58     begin
59         i_c <= i_r;
60         done_o <= '0';
61         case state is
62             when IDLE =>
63                 i_c <= 0;
64                 z_addr_o <= HTR_BASE_ADDR;
65                 wr_o <= '0';
66                 if start_i = '1' then
67                     next_state <= CLEAR_MUL_RES;
68                 end if;
69             when CLEAR_MUL_RES =>
70                 i_c <= i_r + 1;
71                 z_addr_o <= MUL_RES.BASE_ADDR + i_r;
72                 wr_o <= '1';
73                 if i_r = DV*max(M) - 1 then
74                     i_c <= 0;
75                     next_state <= CLEAR_SUM_TMP;
76                 end if;
77             when CLEAR_SUM_TMP =>
78                 i_c <= i_r + 1;
79                 z_addr_o <= SUM_TMP.BASE_ADDR + i_r;
80                 wr_o <= '1';
81                 if i_r = DV*sum(M) - 1 then
82                     i_c <= 0;
83                     next_state <= CLEAR_SIGMA;
84                 end if;
85             when CLEAR_SIGMA =>
86                 i_c <= i_r + 1;
87                 z_addr_o <= SIGMA.BASE_ADDR + i_r;
88                 wr_o <= '1';
89                 if i_r = N0*P - 1 then
90                     i_c <= 0;
91                     next_state <= CLEAR_R;
92                 end if;
93             when CLEAR_R =>
94                 i_c <= i_r + 1;
95                 z_addr_o <= MUL_RES.BASE_ADDR + i_r;
96                 wr_o <= '1';
97                 if i_r = N0*P - 1 then

```



```

98         i_c          <= 0;
99         next_state <= DONE;
100     end if;
101     when DONE =>
102         i_c          <= 0;
103         z_addr_o     <= HTR_BASE_ADDR;
104         wr_o         <= '0';
105         done_o       <= '1';
106         if start_i = '0' then
107             next_state <= IDLE;
108         end if;
109     end case;
110 end process comb;
111
112 end architecture rtl;

```

## Message computation

```

1  library IEEE;
2      use IEEE.std_logic_1164.all;
3      use IEEE.numeric_std.all;
4  library work;
5      use work.system_params.all;
6      use work.matrix_types.all;
7      use work.matrix_mult_functions.all;
8      use work.memory_map.all;
9
10 entity comp_message is
11     port(
12         -- control signals
13         clk_i:      in  std_logic;
14         rst_n_i:    in  std_logic;
15         start_i:    in  std_logic;
16         -- to memory
17         a_i:        in  pos;
18         a_addr_o:   out addr;
19         b_i:        in  pos;
20         b_addr_o:   out addr;
21         z_o:        out pos;
22         z_addr_o:   out addr;
23         wr_o:       out std_logic;
24         -- report back when done
25         done_o:     out std_logic
26     );
27 end entity comp_message;
28
29 architecture rtl of comp_message is
30
31     type state_t is (IDLE, BUSY, DONE);
32     signal state: state_t;
33     signal next_state: state_t;
34
35     signal i_c, i_r: integer;
36
37 begin
38
39     a_addr_o <= X_BASE_ADDR + i_r;
40     b_addr_o <= E_BASE_ADDR + i_r;
41     z_o      <= a_i xor b_i;

```

```

42     z_addr_o    <=  UBASE_ADDR + i_r;
43
44     seq: process(clk_i, rst_n_i)
45     begin
46         if rst_n_i = '0' then
47             state <= IDLE;
48         elsif rising_edge(clk_i) then
49             state <= next_state;
50         end if;
51     end process seq;
52     comb: process(state, start_i, i_r)
53     begin
54         next_state <= state;
55         i_c        <= 0;
56         wr_o        <= '0';
57         done_o       <= '0';
58         case (state) is
59             when IDLE =>
60                 if start_i = '1' then
61                     next_state <= BUSY;
62                 end if;
63             when BUSY =>
64                 i_c <= i_r + 1;
65                 wr_o <= '1';
66                 if i_r = N0*(P-1) - 1 then
67                     next_state <= DONE;
68                 end if;
69             when DONE =>
70                 done_o <= '1';
71                 if start_i = '0' then
72                     next_state <= IDLE;
73                 end if;
74             end case;
75     end process comb;
76
77 end architecture rtl;

```

## A.5 Top module

```

1  library IEEE;
2      use IEEE.std_logic_1164.all;
3      use IEEE.numeric_std.all;
4  library work;
5      use work.system_params.all;
6      use work.matrix_types.all;
7      use work.matrix_mult_functions.all;
8
9  entity top is
10     port(
11         -- control signals
12         clk_i:    in  std_logic;
13         rst_n_i:  in  std_logic;
14         start_i:  in  std_logic;
15         -- to memory
16         a_i:      in  pos;
17         a_addr_o: out  addr;
18         b_i:      in  pos;
19         b_addr_o: out  addr;

```

```

20         z_o:          out pos;
21         z_addr_o:     out addr;
22         wr_o:         out std_logic;
23         -- report back when done
24         failure_o:    out std_logic;
25         done_o:       out std_logic
26     );
27 end entity top;
28
29 architecture rtl of top is
30
31     component key_reconstruction is
32     port (
33         -- control signals
34         clk_i:         in  std_logic;
35         rst_n_i:       in  std_logic;
36         start_i:       in  std_logic;
37         -- data, addresses and controls to memory
38         a_i:           in  pos;
39         a_addr_o:      out addr;
40         b_i:           in  pos;
41         b_addr_o:      out addr;
42         z_o:           out pos;
43         z_addr_o:      out addr;
44         wr_o:          out std_logic;
45         -- report back once done
46         key_rec_done_o: out std_logic
47     );
48 end component key_reconstruction;
49 signal kr_start:      std_logic;
50 signal kr_a_addr:     addr;
51 signal kr_b_addr:     addr;
52 signal kr_z:          pos;
53 signal kr_z_addr:     addr;
54 signal kr_wr:         std_logic;
55 signal kr_done:       std_logic;
56
57 component vector_by_circulant is
58 port(
59     -- control signals
60     clk_i:         in  std_logic;
61     rst_n_i:       in  std_logic;
62     start_i:       in  std_logic;
63     binary_i:      in  std_logic;
64     -- function arguments in (not latched)
65     a_base_i:      in  addr;
66     b_base_i:      in  addr;
67     b_weight:      in  integer;
68     z_base_i:      in  addr;
69     op:            in  std_logic; -- '0' -> z=aB; '1' -> z=Ba
70     -- data, addresses and controls to memory
71     a_i:           in  pos;
72     a_addr_o:      out addr;
73     b_i:           in  pos;
74     b_addr_o:      out addr;
75     z_o:           out pos;
76     z_addr_o:      out addr;
77     wr_o:          out std_logic;
78     done_o:        out std_logic
79 );
80 end component vector_by_circulant;
81 signal vc_start:      std_logic;

```

```

82  signal  vc_binary:      std_logic;
83  signal  vc_vec_base:    addr;
84  signal  vc_blk_base:    addr;
85  signal  vc_blk_weight:  natural;
86  signal  vc_res_base:    addr;
87  signal  vc_op:          std_logic;
88  signal  vc_a_addr:      addr;
89  signal  vc_b_addr:      addr;
90  signal  vc_z:           pos;
91  signal  vc_z_addr:      addr;
92  signal  vc_wr:          std_logic;
93  signal  vc_done:        std_logic;
94
95  component x_by_Ltr is
96      port (
97          -- control signals
98          clk_i:          in  std_logic;
99          rst_n_i:        in  std_logic;
100         start_i:         in  std_logic;
101         -- controls to multiplication core
102         vector_base_o:   out addr;
103         block_base_o:    out addr;
104         result_base_o:   out addr;
105         start_mul_o:     out std_logic;
106         mul_done_i:      in  std_logic;
107         -- report back
108         done_o:          out std_logic
109     );
110 end component x_by_Ltr;
111 signal  xl_start:        std_logic;
112 signal  xl_vec_base:     addr;
113 signal  xl_blk_base:     addr;
114 signal  xl_res_base:     addr;
115 signal  xl_start_mul:    std_logic;
116 signal  xl_done:         std_logic;
117
118 component Htr_by_str is
119     port (
120         -- control signals
121         clk_i:          in  std_logic;
122         rst_n_i:        in  std_logic;
123         start_i:         in  std_logic;
124         -- controls to multiplication core
125         vector_base_o:   out addr;
126         block_base_o:    out addr;
127         result_base_o:   out addr;
128         start_mul_o:     out std_logic;
129         mul_done_i:      in  std_logic;
130         -- report back
131         done_o:          out std_logic
132     );
133 end component Htr_by_str;
134 signal  hs_start:        std_logic;
135 signal  hs_vec_base:     addr;
136 signal  hs_blk_base:     addr;
137 signal  hs_res_base:     addr;
138 signal  hs_start_mul:    std_logic;
139 signal  hs_done:         std_logic;
140
141 component Qtr_by_sigmatr is
142     port (
143         -- control signals

```

```

144         clk_i:          in  std_logic;
145         rst_n_i:        in  std_logic;
146         start_i:        in  std_logic;
147         -- controls to multiplication core
148         vector_base_o:  out  addr;
149         block_base_o:   out  addr;
150         block_weight_o: out  natural;
151         result_base_o:  out  addr;
152         start_mul_o:    out  std_logic;
153         mul_done_i:     in  std_logic;
154         -- report back
155         done_o:         out  std_logic
156     );
157 end component Qtr_by_sigmatr;
158 signal qs_start:      std_logic;
159 signal qs_vec_base:   addr;
160 signal qs_blk_base:   addr;
161 signal qs_blk_weight: natural;
162 signal qs_res_base:   addr;
163 signal qs_start_mul:  std_logic;
164 signal qs_done:       std_logic;
165
166 component find_max is
167     port(
168         -- control signals
169         clk_i:      in  std_logic;
170         rst_n_i:    in  std_logic;
171         start_i:    in  std_logic;
172         -- to memory
173         a_i:        in  pos;
174         a_addr_o:    out addr;
175         b_i:        in  pos;
176         b_addr_o:    out addr;
177         z_o:        out pos;
178         z_addr_o:    out addr;
179         wr_o:       out std_logic;
180         -- report back when done
181         max_idx_o:   out n_array(15 downto 0);
182         done_o:      out std_logic
183     );
184 end component find_max;
185 signal fm_start:      std_logic;
186 signal fm_a_addr:     addr;
187 signal fm_b_addr:     addr;
188 signal fm_z:          pos;
189 signal fm_z_addr:     addr;
190 signal fm_wr:         std_logic;
191 signal fm_max_idx:    n_array(15 downto 0);
192 signal fm_done:       std_logic;
193 signal max_idx_c:     n_array(15 downto 0);
194 signal max_idx_r:     n_array(15 downto 0);
195
196 component vector_plus_rows is
197     port (
198         -- control signals
199         clk_i:      in  std_logic;
200         rst_n_i:    in  std_logic;
201         start_i:    in  std_logic;
202         -- row indexes
203         row_idx_i:  in  n_array(15 downto 0);
204         -- to memory
205         a_i:        in  pos;

```

```

206         a_addr_o:    out addr;
207         b_i:         in  pos;
208         b_addr_o:    out addr;
209         z_o:         out pos;
210         z_addr_o:    out addr;
211         wr_o:        out std_logic;
212         -- report back when done
213         done_o:      out std_logic
214     );
215 end component vector_plus_rows;
216 signal vr_start:    std_logic;
217 signal vr_a_addr:   addr;
218 signal vr_b_addr:   addr;
219 signal vr_z:        pos;
220 signal vr_z_addr:   addr;
221 signal vr_wr:       std_logic;
222 signal vr_done:     std_logic;
223
224 component e_by_Htr is
225     port (
226         -- control signals
227         clk_i:       in  std_logic;
228         rst_n_i:     in  std_logic;
229         start_i:     in  std_logic;
230         -- controls to multiplication core
231         vector_base_o: out addr;
232         block_base_o:  out addr;
233         result_base_o: out addr;
234         start_mul_o:   out std_logic;
235         mul_done_i:    in  std_logic;
236         -- report back
237         done_o:       out std_logic
238     );
239 end component e_by_Htr;
240 signal eh_start:    std_logic;
241 signal eh_vec_base: addr;
242 signal eh_blk_base: addr;
243 signal eh_res_base: addr;
244 signal eh_start_mul: std_logic;
245 signal eh_done:     std_logic;
246
247 component null_syndrome is
248     port(
249         -- control signals
250         clk_i:       in  std_logic;
251         rst_n_i:     in  std_logic;
252         start_i:     in  std_logic;
253         -- to memory
254         a_i:         in  pos;
255         a_addr_o:    out addr;
256         b_i:         in  pos;
257         b_addr_o:    out addr;
258         z_o:         out pos;
259         z_addr_o:    out addr;
260         wr_o:        out std_logic;
261         -- report back when done
262         null_syn_o:  out std_logic;
263         done_o:      out std_logic
264     );
265 end component null_syndrome;
266 signal ns_start:    std_logic;
267 signal ns_a_addr:   addr;

```

```

268 signal ns_b_addr:      addr;
269 signal ns_z:          pos;
270 signal ns_z_addr:     addr;
271 signal ns_wr:         std_logic;
272 signal ns_null_syn:   std_logic;
273 signal ns_done:       std_logic;
274
275 component clear_temp is
276     port(
277         -- control signals
278         clk_i:        in  std_logic;
279         rst_n_i:       in  std_logic;
280         start_i:       in  std_logic;
281         -- to memory
282         a_i:          in  pos;
283         a_addr_o:     out addr;
284         b_i:          in  pos;
285         b_addr_o:     out addr;
286         z_o:          out pos;
287         z_addr_o:     out addr;
288         wr_o:         out std_logic;
289         -- report back when done
290         done_o:       out std_logic
291     );
292 end component clear_temp;
293 signal ct_start:      std_logic;
294 signal ct_a_addr:     addr;
295 signal ct_b_addr:     addr;
296 signal ct_z:          pos;
297 signal ct_z_addr:     addr;
298 signal ct_wr:         std_logic;
299 signal ct_null_syn:   std_logic;
300 signal ct_done:       std_logic;
301
302 component comp_message is
303     port(
304         -- control signals
305         clk_i:        in  std_logic;
306         rst_n_i:       in  std_logic;
307         start_i:       in  std_logic;
308         -- to memory
309         a_i:          in  pos;
310         a_addr_o:     out addr;
311         b_i:          in  pos;
312         b_addr_o:     out addr;
313         z_o:          out pos;
314         z_addr_o:     out addr;
315         wr_o:         out std_logic;
316         -- report back when done
317         done_o:       out std_logic
318     );
319 end component comp_message;
320 signal cm_start:      std_logic;
321 signal cm_a_addr:     addr;
322 signal cm_b_addr:     addr;
323 signal cm_z:          pos;
324 signal cm_z_addr:     addr;
325 signal cm_wr:         std_logic;
326 signal cm_done:       std_logic;
327
328 type state_t is (
329     IDLE,

```

```

330     COMPUTE_LTR,
331     COMPUTE_INITIAL_SYNDROME,
332     COMPUTE_SIGMA,
333     COMPUTER,
334     FIND_B,
335     COMPUTE_ERROR,
336     COMPUTE_SYNDROME,
337     CHECK_SYNDROME,
338     CLEAR_TEMP_AND_LOOP,
339     COMPUTE_MESSAGE,
340     CLEAR_TEMP_AND_RETURN,
341     DONE
342 );
343 signal state: state_t;
344 signal next_state: state_t;
345
346 signal l_c: integer;
347 signal l_r: integer;
348
349 begin
350
351     KR: key_reconstruction
352     port map (
353         clk_i      => clk_i ,
354         rst_n_i     => rst_n_i ,
355         start_i     => kr_start ,
356         a_i         => a_i ,
357         a_addr_o    => kr_a_addr ,
358         b_i         => b_i ,
359         b_addr_o    => kr_b_addr ,
360         z_o         => kr_z ,
361         z_addr_o    => kr_z_addr ,
362         wr_o        => kr_wr ,
363         key_rec_done_o => kr_done
364     );
365
366     VC: vector_by_circulant
367     port map (
368         clk_i      => clk_i ,
369         rst_n_i     => rst_n_i ,
370         start_i     => vc_start ,
371         binary_i    => vc_binary ,
372         a_base_i    => vc_vec_base ,
373         b_base_i    => vc_blk_base ,
374         b_weight    => vc_blk_weight ,
375         z_base_i    => vc_res_base ,
376         op          => vc_op ,
377         a_i         => a_i ,
378         a_addr_o    => vc_a_addr ,
379         b_i         => b_i ,
380         b_addr_o    => vc_b_addr ,
381         z_o         => vc_z ,
382         z_addr_o    => vc_z_addr ,
383         wr_o        => vc_wr ,
384         done_o      => vc_done
385     );
386
387     XL: x_by_Ltr
388     port map (
389         clk_i      => clk_i ,
390         rst_n_i     => rst_n_i ,
391         start_i     => xl_start ,

```



```

392         vector_base_o    => xl_vec_base ,
393         block_base_o     => xl_blk_base ,
394         result_base_o    => xl_res_base ,
395         start_mul_o      => xl_start_mul ,
396         mul_done_i       => vc_done ,
397         done_o           => xl_done
398     );
399
400 HS: Htr_by_str
401     port map (
402         clk_i             => clk_i ,
403         rst_n_i           => rst_n_i ,
404         start_i           => hs_start ,
405         vector_base_o     => hs_vec_base ,
406         block_base_o     => hs_blk_base ,
407         result_base_o    => hs_res_base ,
408         start_mul_o      => hs_start_mul ,
409         mul_done_i       => vc_done ,
410         done_o           => hs_done
411     );
412
413 QS: Qtr_by_sigmatr
414     port map (
415         clk_i             => clk_i ,
416         rst_n_i           => rst_n_i ,
417         start_i           => qs_start ,
418         vector_base_o     => qs_vec_base ,
419         block_base_o     => qs_blk_base ,
420         block_weight_o    => qs_blk_weight ,
421         result_base_o    => qs_res_base ,
422         start_mul_o      => qs_start_mul ,
423         mul_done_i       => vc_done ,
424         done_o           => qs_done
425     );
426
427 FM: find_max
428     port map (
429         clk_i             => clk_i ,
430         rst_n_i           => rst_n_i ,
431         start_i           => fm_start ,
432         a_i               => a_i ,
433         a_addr_o          => fm_a_addr ,
434         b_i               => b_i ,
435         b_addr_o          => fm_b_addr ,
436         z_o               => fm_z ,
437         z_addr_o          => fm_z_addr ,
438         wr_o              => fm_wr ,
439         max_idx_o         => fm_max_idx ,
440         done_o            => fm_done
441     );
442
443 VR: vector_plus_rows
444     port map (
445         clk_i             => clk_i ,
446         rst_n_i           => rst_n_i ,
447         start_i           => vr_start ,
448         row_idx_i         => max_idx_r ,
449         a_i               => a_i ,
450         a_addr_o          => vr_a_addr ,
451         b_i               => b_i ,
452         b_addr_o          => vr_b_addr ,
453         z_o               => vr_z ,

```

```

454         z_addr_o      => vr_z_addr ,
455         wr_o           => vr_wr ,
456         done_o         => vr_done
457     );
458
459 EH: e_by_Htr
460     port map (
461         clk_i           => clk_i ,
462         rst_n_i         => rst_n_i ,
463         start_i         => eh_start ,
464         vector_base_o   => eh_vec_base ,
465         block_base_o    => eh_blk_base ,
466         result_base_o   => eh_res_base ,
467         start_mul_o     => eh_start_mul ,
468         mul_done_i      => vc_done ,
469         done_o          => eh_done
470     );
471
472 NS: null_syndrome
473     port map (
474         clk_i           => clk_i ,
475         rst_n_i         => rst_n_i ,
476         start_i         => ns_start ,
477         a_i             => a_i ,
478         a_addr_o        => ns_a_addr ,
479         b_i             => b_i ,
480         b_addr_o        => ns_b_addr ,
481         z_o             => ns_z ,
482         z_addr_o        => ns_z_addr ,
483         wr_o            => ns_wr ,
484         null_syn_o      => ns_null_syn ,
485         done_o          => ns_done
486     );
487
488 CT: clear_temp
489     port map (
490         clk_i           => clk_i ,
491         rst_n_i         => rst_n_i ,
492         start_i         => ct_start ,
493         a_i             => a_i ,
494         a_addr_o        => ct_a_addr ,
495         b_i             => b_i ,
496         b_addr_o        => ct_b_addr ,
497         z_o             => ct_z ,
498         z_addr_o        => ct_z_addr ,
499         wr_o            => ct_wr ,
500         done_o          => ct_done
501     );
502
503 CM: comp_message
504     port map (
505         clk_i           => clk_i ,
506         rst_n_i         => rst_n_i ,
507         start_i         => cm_start ,
508         a_i             => a_i ,
509         a_addr_o        => cm_a_addr ,
510         b_i             => b_i ,
511         b_addr_o        => cm_b_addr ,
512         z_o             => cm_z ,
513         z_addr_o        => cm_z_addr ,
514         wr_o            => cm_wr ,
515         done_o          => cm_done

```

```

516         );
517
518     seq: process (clk_i, rst_n_i)
519     begin
520         if rst_n_i = '0' then
521             state <= IDLE;
522             max_idx_r <= (others => 0);
523             l_r <= 0;
524         elsif rising_edge(clk_i) then
525             state <= next_state;
526             max_idx_r <= max_idx_c;
527             l_r <= l_c;
528         end if;
529     end process seq;
530     -- TODO: put everything in sensitivity list
531     comb: process (all)
532     --state, start_i,
533     --    kr_a_addr, kr_b_addr, kr_z, kr_z_addr, kr_wr, kr_done,
534     --    vc_a_addr, vc_b_addr, vc_z, vc_z_addr, vc_wr,
535     --    xl_start_mul, xl_vec_base, xl_blk_base, xl_res_base, xl_done,
536     --    hs_start_mul, hs_vec_base, hs_blk_base, hs_res_base, hs_done,
537     --    qs_start_mul, qs_vec_base, qs_blk_base, qs_res_base, qs_done,
538     --    fm_a_addr, fm_b_addr, fm_z, fm_z_addr, fm_wr, fm_max_idx, fm_done,
539     --    vr_a_addr, vr_b_addr, vr_z, vr_z_addr, vr_wr, vr_done,
540     --    eh_start_mul, eh_vec_base, eh_blk_base, eh_res_base, eh_done)
541     begin
542         next_state <= state;
543         a_addr_o <= to_unsigned(0, address_bits);
544         b_addr_o <= to_unsigned(0, address_bits);
545         z_o <= to_unsigned(0, position_bits);
546         z_addr_o <= to_unsigned(0, address_bits);
547         wr_o <= '0';
548         failure_o <= '0';
549         done_o <= '0';
550         kr_start <= '0';
551         vc_start <= '0';
552         vc_binary <= '0';
553         vc_vec_base <= to_unsigned(0, address_bits);
554         vc_blk_base <= to_unsigned(0, address_bits);
555         vc_blk_weight <= 0;
556         vc_res_base <= to_unsigned(0, address_bits);
557         vc_op <= '0';
558         xl_start <= '0';
559         hs_start <= '0';
560         qs_start <= '0';
561         fm_start <= '0';
562         max_idx_c <= max_idx_r;
563         vr_start <= '0';
564         eh_start <= '0';
565         ns_start <= '0';
566         ct_start <= '0';
567         cm_start <= '0';
568         l_c <= l_r;
569         case state is
570             when IDLE =>
571                 l_c <= 0;
572                 if start_i = '1' then
573                     next_state <= COMPUTELTR;
574                 end if;
575             when COMPUTELTR =>
576                 a_addr_o <= kr_a_addr;
577                 b_addr_o <= kr_b_addr;

```

```

578         z_o          <= kr_z;
579         z_addr_o     <= kr_z_addr;
580         wr_o         <= kr_wr;
581         kr_start     <= '1';
582         if kr_done = '1' then
583             next_state <= COMPUTE_INITIAL_SYNDROME;
584         end if;
585     when COMPUTE_INITIAL_SYNDROME =>
586         a_addr_o     <= vc_a_addr;
587         b_addr_o     <= vc_b_addr;
588         z_o          <= vc_z;
589         z_addr_o     <= vc_z_addr;
590         wr_o         <= vc_wr;
591         vc_start     <= xl_start_mul;
592         vc_binary    <= '1';
593         vc_vec_base  <= xl_vec_base;
594         vc_blk_base  <= xl_blk_base;
595         vc_blk_weight <= sum(M)*DV;
596         vc_res_base  <= xl_res_base;
597         vc_op        <= '0';
598         xl_start     <= '1';
599         if xl_done = '1' then
600             next_state <= COMPUTE_SIGMA;
601         end if;
602     when COMPUTE_SIGMA =>
603         a_addr_o     <= vc_a_addr;
604         b_addr_o     <= vc_b_addr;
605         z_o          <= vc_z;
606         z_addr_o     <= vc_z_addr;
607         wr_o         <= vc_wr;
608         vc_start     <= hs_start_mul;
609         vc_binary    <= '0';
610         vc_vec_base  <= hs_vec_base;
611         vc_blk_base  <= hs_blk_base;
612         vc_blk_weight <= DV;
613         vc_res_base  <= hs_res_base;
614         vc_op        <= '1';
615         hs_start     <= '1';
616         if hs_done = '1' then
617             next_state <= COMPUTER;
618         end if;
619     when COMPUTER =>
620         a_addr_o     <= vc_a_addr;
621         b_addr_o     <= vc_b_addr;
622         z_o          <= vc_z;
623         z_addr_o     <= vc_z_addr;
624         wr_o         <= vc_wr;
625         vc_start     <= qs_start_mul;
626         vc_binary    <= '0';
627         vc_vec_base  <= qs_vec_base;
628         vc_blk_base  <= qs_blk_base;
629         vc_blk_weight <= qs_blk_weight + 1;
630         vc_res_base  <= qs_res_base;
631         vc_op        <= '1';
632         qs_start     <= '1';
633         if qs_done = '1' then
634             next_state <= FIND_B;
635         end if;
636     when FIND_B =>
637         a_addr_o     <= fm_a_addr;
638         b_addr_o     <= fm_b_addr;
639         z_o          <= fm_z;

```

```

640         z_addr_o      <= fm_z_addr;
641         wr_o           <= fm_wr;
642         fm_start       <= '1';
643         if fm_done = '1' then
644             max_idx_c  <= fm_max_idx;
645             next_state <= COMPUTEERROR;
646         end if;
647     when COMPUTEERROR =>
648         a_addr_o      <= vr_a_addr;
649         b_addr_o      <= vr_b_addr;
650         z_o           <= vr_z;
651         z_addr_o      <= vr_z_addr;
652         wr_o           <= vr_wr;
653         vr_start       <= '1';
654         if vr_done = '1' then
655             next_state <= COMPUTESYNDROME;
656         end if;
657     when COMPUTESYNDROME =>
658         a_addr_o      <= vc_a_addr;
659         b_addr_o      <= vc_b_addr;
660         z_o           <= vc_z;
661         z_addr_o      <= vc_z_addr;
662         wr_o           <= vc_wr;
663         vc_start       <= eh_start_mul;
664         vc_binary      <= '1';
665         vc_vec_base    <= eh_vec_base;
666         vc_blk_base    <= eh_blk_base;
667         vc_blk_weight  <= sum(M)*DV;
668         vc_res_base    <= eh_res_base;
669         vc_op          <= '0';
670         eh_start       <= '1';
671         if eh_done = '1' then
672             next_state <= CHECKSYNDROME;
673         end if;
674     when CHECKSYNDROME =>
675         a_addr_o      <= ns_a_addr;
676         b_addr_o      <= ns_b_addr;
677         z_o           <= ns_z;
678         z_addr_o      <= ns_z_addr;
679         wr_o           <= ns_wr;
680         ns_start       <= '1';
681         if ns_done = '1' then
682             if ns_null_syn = '1' or l_r >= 20 then
683                 next_state <= COMPUTEMESSAGE;
684             else
685                 next_state <= CLEAR_TEMP_AND_LOOP;
686             end if;
687         end if;
688     when CLEAR_TEMP_AND_LOOP =>
689         a_addr_o      <= ct_a_addr;
690         b_addr_o      <= ct_b_addr;
691         z_o           <= ct_z;
692         z_addr_o      <= ct_z_addr;
693         wr_o           <= ct_wr;
694         ct_start       <= '1';
695         if ct_done = '1' then
696             l_c        <= l_r + 1;
697             next_state <= COMPUTESIGMA;
698         end if;
699     when COMPUTEMESSAGE =>
700         a_addr_o      <= cm_a_addr;
701         b_addr_o      <= cm_b_addr;

```

```

702         z_o          <= cm_z;
703         z_addr_o     <= cm_z_addr;
704         wr_o         <= cm_wr;
705         cm_start     <= '1';
706         if cm_done = '1' then
707             next_state <= CLEAR_TEMP_AND_RETURN;
708         end if;
709     when CLEAR_TEMP_AND_RETURN =>
710         a_addr_o     <= ct_a_addr;
711         b_addr_o     <= ct_b_addr;
712         z_o          <= ct_z;
713         z_addr_o     <= ct_z_addr;
714         wr_o         <= ct_wr;
715         ct_start     <= '1';
716         if ct_done = '1' then
717             next_state <= DONE;
718         end if;
719     when DONE =>
720         done_o <= '1';
721         if l_r >= 20 then
722             failure_o <= '1';
723         end if;
724         if start_i = '0' then
725             next_state <= IDLE;
726         end if;
727     when others =>
728         null;
729     end case;
730 end process comb;
731
732 end architecture rtl;

```

# Bibliography

- [1] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini, “LEDAkem: a post-quantum key encapsulation mechanism based on QC-LDPC codes”, arXiv:1801.08867v1 [cs.CR] 26 Jan 2018
- [2] [https://en.wikipedia.org/wiki/Insertion\\_sort](https://en.wikipedia.org/wiki/Insertion_sort), rev. 11 May 2018
- [3] James F. Kurose, Keith W. Ross, “Computer Networking: a top-down approach”, Pearson 2013
- [4] Michele Elia, “An Introduction to Classic Cryptography”, Aracne Editrice 2018
- [5] R. Rivest, A. Shamir, L. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”, Communications of the ACM, February 1978, pp. 120-126
- [6] Peter Shor, “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer”, arXiv:quant-ph/9508027v2 25 Jan 1996
- [7] Gordon E. Moore, “Cramming more components onto integrated circuits”, Electronics, Volume 38, Number 8, April 19, 1965
- [8] Robert J. McEliece, “A Public-Key Cryptosystem Based On Algebraic Coding Theory”, DSN Progress Report 42-44, January-February 1978, pp. 114-116
- [9] David A. Patterson, Garth Gibson, Randy H. Katz, “A Case for Redundant Arrays of Inexpensive Disks (RAID)”, ACM 1988
- [10] [https://en.wikipedia.org/wiki/Merge\\_sort](https://en.wikipedia.org/wiki/Merge_sort), rev. 25 November 2018