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

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.

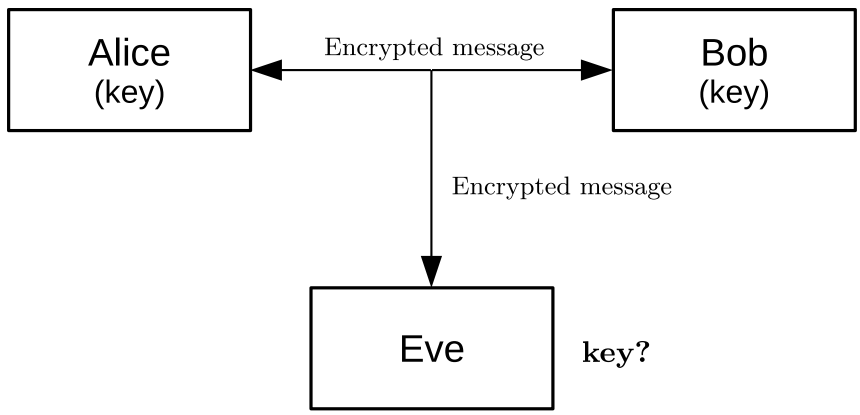


Figure 1.1: Alice, Bob and Eve

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

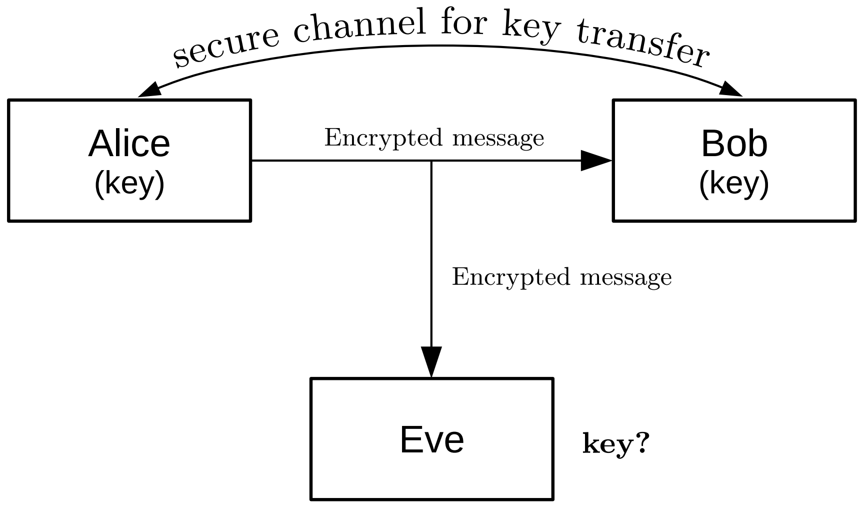


Figure 1.2: Secure channel for key transmission

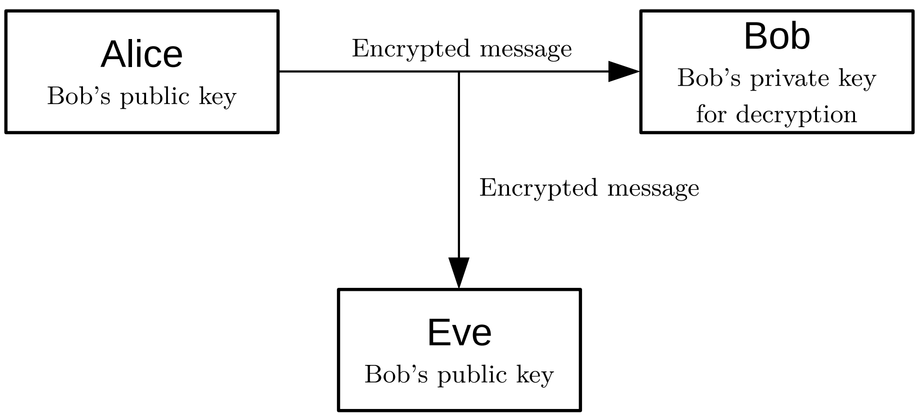


Figure 1.3: Asymmetric key not requiring a secure channel

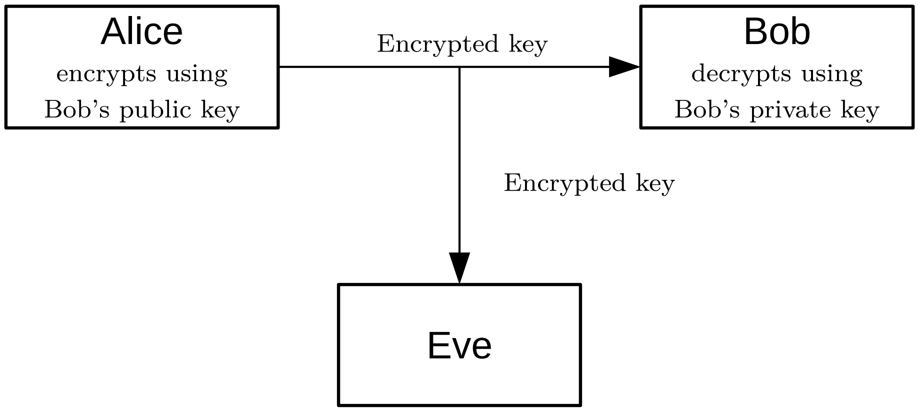


Figure 1.4: Asymmetric cypher creating a secure channel for a symmetric key

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 **H** and **Q**:

$$\mathbf{H} = \begin{bmatrix} \mathbf{H_0} & | & \mathbf{H_1} & | & \cdots & | & \mathbf{H_{n_0-1}} \end{bmatrix} \tag{2.1}$$

$$\mathbf{Q} = \begin{bmatrix} \mathbf{Q_{0,0}} & | & \mathbf{Q_{0,1}} & | & \cdots & | & \mathbf{Q_{0,n_0-1}} \\ \mathbf{Q_{1,0}} & | & \mathbf{Q_{1,1}} & | & \cdots & | & \mathbf{Q_{1,n_0-1}} \\ \vdots & | & \vdots & | & \ddots & | & \vdots \\ \mathbf{Q_{n_0-1,0}} & | & \mathbf{Q_{n_0-1,1}} & | & \cdots & | & \mathbf{Q_{n_0-1,n_0-1}} \end{bmatrix} \tag{2.2}$$

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 **H** have weight (number of set elements) d_v, with a standard choice being 17, while blocks of **Q** have a weight according to the following map (which is, by the way, circulant as well):

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

where m_i are again small integer values.

From matrices **H** and **Q** a new matrix **L** is obtained as:

$$\mathbf{L} = \mathbf{HQ} = \begin{bmatrix} \mathbf{L_0} & | & \mathbf{L_1} & | & \cdots & | & \mathbf{L_{n_0-1}} \end{bmatrix}$$

Given a proper choice of parameters d_v and $\underline{m} = [m_0, m_1, ..., m_{n_0-1}]$ 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:

$$\mathbf{M} = \mathbf{L_{n_0-1}^{-1}L} = \begin{bmatrix}\mathbf{M_0} & | & \mathbf{M_1} & | & \cdots & | & \mathbf{M_{n_0-2}} & | & \mathbf{I_p}\end{bmatrix} = \begin{bmatrix}\mathbf{M_l}|\mathbf{I_p}\end{bmatrix}$$

The generator matrix is then obtained as:

$$\mathbf{G'} = \begin{bmatrix}\mathbf{I_{p(n_0-1)}} & | & \mathbf{M_l^T}\end{bmatrix}$$

with $\mathbf{M_l^T}$ being the transpose of $\mathbf{M_l}$. 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 $\mathbf{M_l}$.

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 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

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

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.

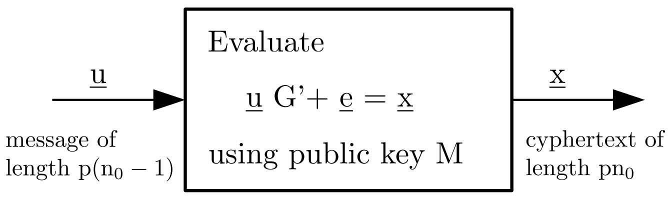


Figure 2.1: Encryption in LEDAcrypt

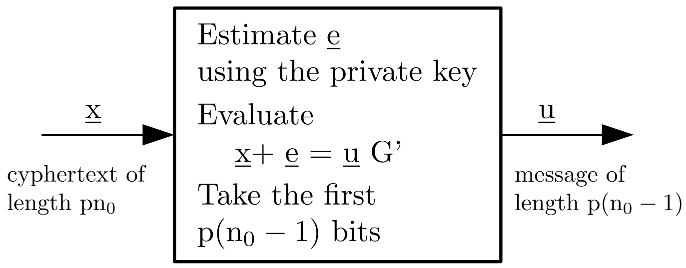


Figure 2.2: Decryption in LEDAcrypt

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 **H** and **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^{th} row of **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:

$$\underline{u}\mathbf{G}' = \underline{x} + \underline{e}^{(l)}$$

$$\underline{u} = (\underline{x} + \underline{e}^{(l)})[0 : p(n0-1)-1]$$

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.

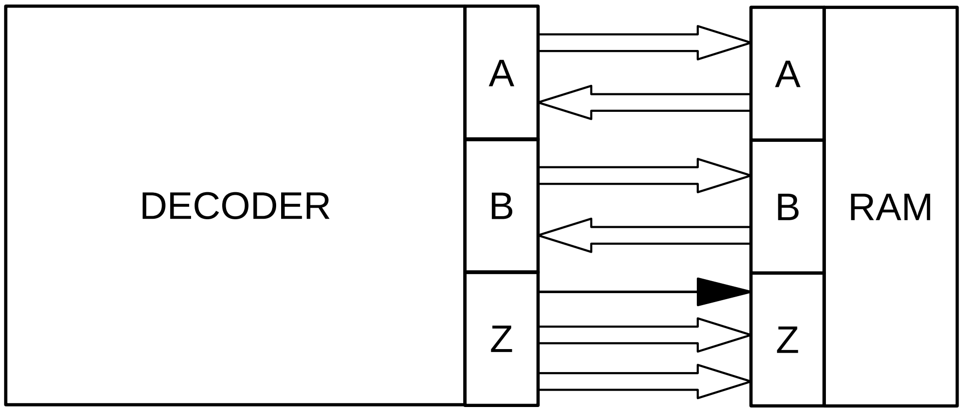


Figure 3.1: Decoder and memory

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

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

Handling traditional matrix multiplication, with $\mathbf{H}$ having size 27779×55558 and $\mathbf{Q}$ having size 55558×55558 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:

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

$$\mathbf{B} = \begin{bmatrix} b_0 & b_1 & \cdots & b_{n-1} \end{bmatrix}$$
$$\mathbf{AB} = \begin{bmatrix} a_0 + b_0 & a_0 + b_1 & \cdots & a_1 + b_0 & \cdots & a_{m-1} + b_{n-1} \end{bmatrix}$$

with a_i being the position of the i^{th} set element of $\mathbf{A}$ and b_j being the position of the j^{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:

$$a_i < p$$

$$b_j < p$$

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

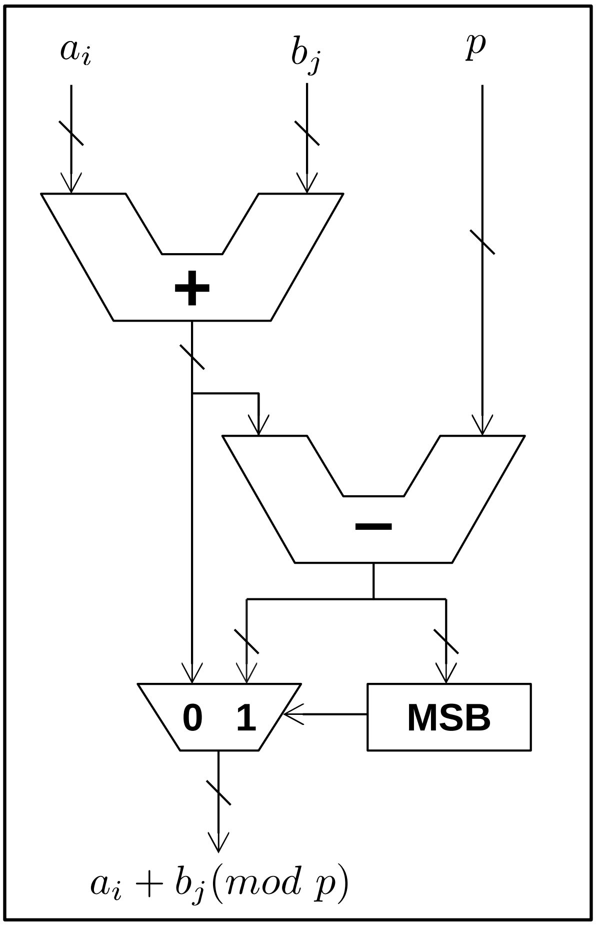
This means that there are only two possible solutions to $a_i + b_j \pmod p$: either $a_i + b_j$ or $a_i + b_j - p$. The second value is computed in the same cycle as the first one and a simple comparator then selects the proper result: this is easily done by checking the sign of $a_i + b_j - p$, as 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

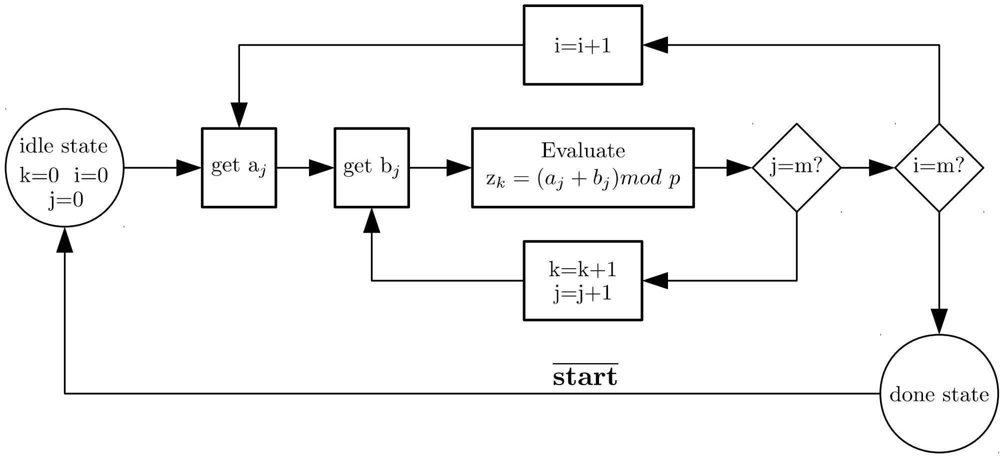


Figure 4.2: Multiplication between two circulant blocks

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^{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^{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^{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 intersted 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^{th} row of $\mathbf{Q}^T$ and the j^{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:

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

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:

$$\sum_{m=0}^{n_0-1} (\mathbf{Q}_{x,m}\mathbf{H}_m) = \sum_{m=0}^{n0-1} \left(\sum_{n=mp}^{(m+1)p-1} q_{i,n}h_{n,j} \right) \quad i \in [xp, (x+1)p-1]$$

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

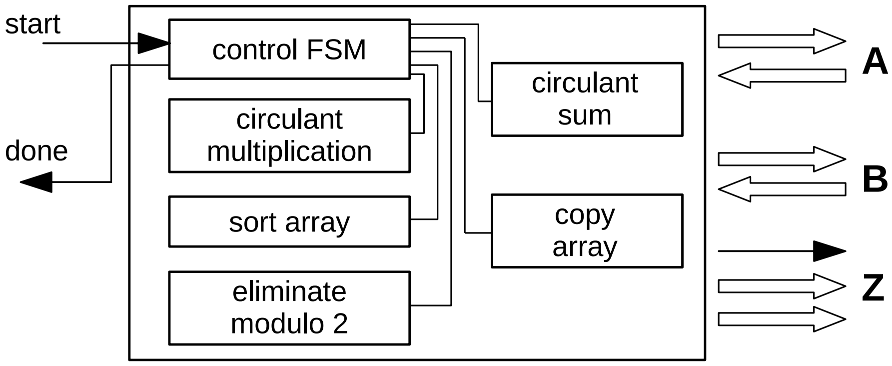


Figure 4.3: Module computing $\mathbf{L}^T = \mathbf{Q}^T\mathbf{H}^T$

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.

$$\underline{a}^T = \begin{bmatrix} a_0 & a_1 & \cdots & a_{p-1} \end{bmatrix}$$

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

$$\underline{c}^T = \begin{bmatrix} c_0 & c_1 & \cdots & c_{p-1} \end{bmatrix}$$

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^{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 wether 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:

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

$$\underline{s} = \begin{bmatrix} s_0 & s_1 & \cdots & s_{p-1} \end{bmatrix}$$

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

$$\underline{x} = \begin{bmatrix} \underline{x}_0 & \underline{x}_1 & \cdots & \underline{x}_{n_0-1} \end{bmatrix} = \begin{bmatrix} x_0 & x_1 & \cdots & x_{p-1} & x_p & \cdots & x_{n_0p-1} \end{bmatrix}$$

$\underline{x}$ 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^{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^{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 σ_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^{th} row of a circulant block $\mathbf{A}$:

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

$$\underline{a}_l^T = \begin{bmatrix} a_{p-l \pmod p} & a_{p-l+1 \pmod p} & a_{p-l+2 \pmod p} & \cdots a_{2p-l-1} \pmod p \end{bmatrix}$$

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 orther 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 minimals. 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.

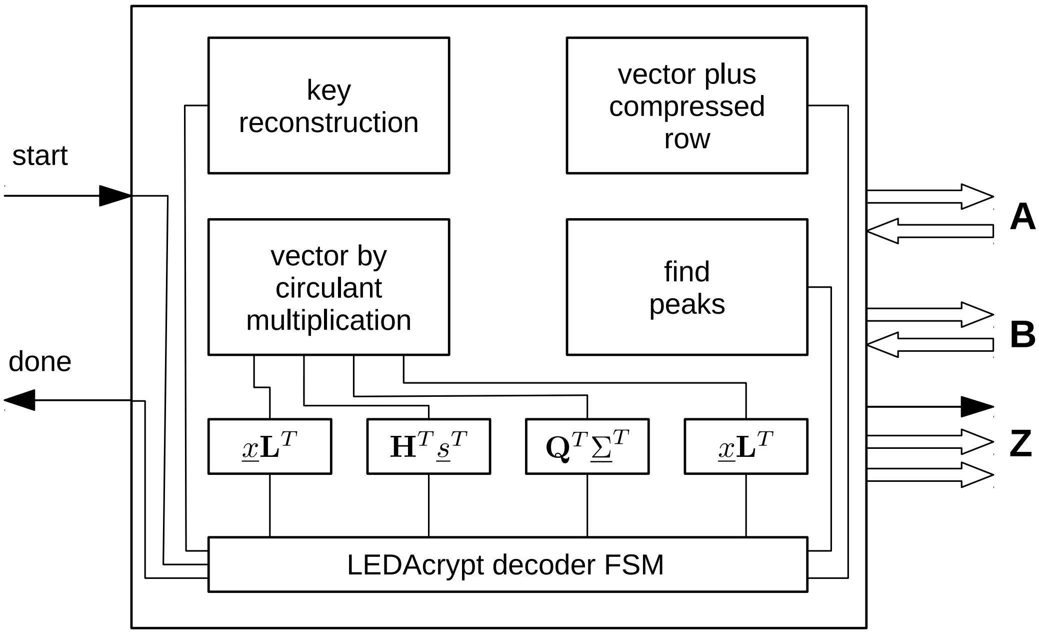


Figure 7.1: Decryption module

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 then 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

```
--------------------------------------------------------------------------------
-- Author:  Flavio Tanese
-- Politecnico di Torino 2018
--------------------------------------------------------------------------------
-- Multiply two sparse circulant binary matrices stored in a memory as the
-- positions of set bits in their first row, and return a "tentative" result
-- (which is unordered and not simplified modulo 2) in another location in
-- memory.
-- Controlling circuitry should take care of keeping all inputs in the
-- "function arguments" section constant until the module reported back,
-- and of ensuring validity of such inputs (no overlapping memory ranges and
-- so on).
-- The "start_i" signal should be kept high until "mult_done_o" goes high,
-- implementing a rudimentary handshake, but this is not strictly required if
-- the control circuitry operates on the same clock.

library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
library work;
    use work.system_params.all;
    use work.matrix_types.all;
    use work.matrix_mult_functions.all;

entity circulant_multiplication is
    port(
        -- control signals
        clk_i:          in  std_logic;
        rst_n_i:        in  std_logic;
        start_i:        in  std_logic;
        -- function arguments in (not latched)
        a_limit_i:      in  natural;    -- number of elements in 1st matrix
        a_base_i:       in  addr;       -- base address of 1st matrix
        b_limit_i:      in  natural;    -- number of elements in 2nd matrix
        b_base_i:       in  addr;       -- base address of 2nd matrix
        z_base_i:       in  addr;       -- base address of result matrix
```

```vhdl
        -- data, addresses and controls to memory
        a_i:            in   pos;        -- operand from 1st matrix
        a_addr_o:       out  addr;       -- address for 1st matrix
        b_i:            in   pos;        -- operand from 2nd matrix
        b_addr_o:       out  addr;       -- address for 2nd matrix
        z_o:            out  pos;        -- Ltr result
        z_addr_o:       out  addr;       -- address for Ltr result
        wr_o:           out  std_logic;  -- write enable for memory
        -- report back once done
        z_limit_o:      out  natural;    -- number of elements in result matrix
        mult_done_o:    out  std_logic   -- high when done
    );
end entity circulant_multiplication;

architecture rtl of circulant_multiplication is

    -- assume inputs are kept constant by higher level state machine so we do
    -- not need to sample them on start

    signal      a_index:            natural;
    signal      b_index:            natural;
    signal      z_index:            natural;
    signal      next_a_index:       natural;
    signal      next_b_index:       natural;
    signal      next_z_index:       natural;

    type state_t is (IDLE, BUSY, DONE);
    signal      state:              state_t;
    signal      next_state:         state_t;

begin

    z_o         <=  a_i + b_i when a_i + b_i < P else
                    (a_i + b_i) mod P;

    a_addr_o    <=  a_base_i + a_index;
    b_addr_o    <=  b_base_i + b_index;
    z_addr_o    <=  z_base_i + z_index;

    -- count number of elements of result matrix, this can be done with a
    -- multiplier but we are not in a hurry and do not want a big footprint.
    -- Using z_index we get that for free!
    z_limit_o   <=  z_index;

    state_comb: process(
        state, start_i, a_limit_i, b_limit_i, next_a_index, next_b_index
    )
    begin
        case state is
            when IDLE =>
                if start_i = '1' then
                    next_state  <=  BUSY;
                else
                    next_state  <=  IDLE;
                end if;
            when BUSY =>
                -- exit this state only once all combinations have been done
                if b_index /= b_limit_i or a_index /= a_limit_i then
                    next_state  <=  BUSY;
                else
                    next_state  <=  DONE;
                end if;
```

```
            when DONE =>
                if start_i = '0' then
                    next_state  <=  IDLE;
                else
                    next_state  <=  DONE;
                end if;
            when others =>
                next_state  <=  IDLE;
        end case;
    end process state_comb;

    state_seq: process(rst_n_i, clk_i)
    begin
        if rst_n_i = '0' then
            state   <=  IDLE;
        elsif rising_edge(clk_i) then
            state   <=  next_state;
        end if;
    end process state_seq;

    output_comb: process(
        state, a_index, b_index, z_index
    )
    begin
        case state is
            when IDLE =>
                next_a_index    <=  0;
                next_b_index    <=  0;
                next_z_index    <=  0;
                wr_o            <=  '0';
                mult_done_o     <=  '0';
            when BUSY =>
                if a_index < a_limit_i then
                    next_a_index    <=  a_index + 1;
                    next_b_index    <=  b_index;
                else
                    next_a_index    <=  0;
                    next_b_index    <=  b_index + 1;
                end if;
                next_z_index    <=  z_index + 1;
                wr_o            <=  '1';
                mult_done_o     <=  '0';
            when DONE =>
                next_a_index    <=  a_index;
                next_b_index    <=  b_index;
                next_z_index    <=  z_index;
                wr_o            <=  '0';
                mult_done_o     <=  '1';
            when others =>  -- behave like IDLE
                next_a_index    <=  0;
                next_b_index    <=  0;
                next_z_index    <=  0;
                wr_o            <=  '0';
                mult_done_o     <=  '0';
        end case;
    end process output_comb;

    output_seq: process(clk_i, rst_n_i)
    begin
        if rst_n_i = '0' then
            a_index     <=  0;
            b_index     <=  0;
```

```
            z_index     <=  0;
        elsif rising_edge(clk_i) then
            a_index     <=  next_a_index;
            b_index     <=  next_b_index;
            z_index     <=  next_z_index;
        end if;
    end process output_seq;

end architecture rtl;
```

A.1.1 Sorting

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
library work;
    use work.system_params.all;
    use work.matrix_types.all;
    use work.matrix_mult_functions.all;

entity sort is
    port(
        -- control logic
        clk_i:          in  std_logic;
        rst_n_i:        in  std_logic;
        start_i:        in  std_logic;
        -- function arguments in (not latched)
        limit_i:        in  natural;    -- number of elements in array to sort
        base_i:         in  addr;       -- base addr of array to sort
        -- data, addresses and controls to memory
        a_i:            in  pos;        -- element A from array
        a_addr_o:       out addr;       -- address of element A
        b_i:            in  pos;        -- element B from array
        b_addr_o:       out addr;       -- address of element B
        z_o:            out pos;        -- element Z to array
        z_addr_o:       out addr;       -- address of element Z
        wr_o:           out std_logic;  -- write enable for memory
        -- report back once done
        sort_done_o:    out std_logic   -- high when done
    );
end entity sort;

architecture rtl of sort is

    -- element A will be overwritten in the array, so we need to sample it in
    -- order to insert it in the right place later on
    signal tmp:         pos;
    signal next_tmp:    pos;

    -- iterators
    signal i:           natural;
    signal j:           natural;
    signal next_i:      natural;
    signal next_j:      natural;

    -- state machine
    type state_t is (
        IDLE, PREP_INNER, CHECK_INNER, LOOP_INNER,
        LAST_INNER, SAVE_0, SAVE_1, DONE
```

```vhdl
    );
    signal  state:      state_t;
    signal  next_state: state_t;

begin

    b_addr_o  <=  base_i + to_unsigned(j, address_bits);

    state_comb: process(state, start_i, next_j, next_i, a_i, b_i)
    begin
        case state is
            when IDLE =>
                if start_i = '1' then
                    next_state  <=  PREP_INNER;
                else
                    next_state  <=  IDLE;
                end if;
            when PREP_INNER =>
                next_state  <=  CHECK_INNER;
            when CHECK_INNER =>
                if j /= 0 and b_i > tmp then
                    next_state  <=  LOOP_INNER;
                elsif j = 0 and b_i > tmp then
                    next_state  <=  LAST_INNER;
                else
                    next_state  <=  SAVE_1;
                end if;
            when LOOP_INNER =>
                if next_j /= 0 and a_i > tmp then
                    next_state  <=  LOOP_INNER;
                elsif next_j = 0 and a_i > tmp then
                    next_state  <=  LAST_INNER;
                else
                    next_state  <=  SAVE_1;
                end if;
            when LAST_INNER =>
                next_state  <=  SAVE_0;
            when SAVE_0 =>
                if i /= limit_i - 1 then
                    next_state  <=  PREP_INNER;
                else
                    next_state  <=  DONE;
                end if;
            when SAVE_1 =>
                if i /= limit_i - 1 then
                    next_state  <=  PREP_INNER;
                else
                    next_state  <=  DONE;
                end if;
            when DONE =>
                if start_i = '0' then
                    next_state  <=  IDLE;
                else
                    next_state  <=  DONE;
                end if;
            when others =>
                next_state  <=  IDLE;
        end case;
    end process state_comb;

    state_seq: process(rst_n_i, clk_i)
    begin
```

```
        if rst_n_i = '0' then
            state   <=  next_state;
        elsif rising_edge(clk_i) then
            state   <=  next_state;
        end if;
    end process state_seq;

    output_comb: process(state, a_i, b_i, tmp, i, j)
    begin
        case state is
            when IDLE =>
                -- internal signals
                next_tmp    <=  tmp;
                next_i      <=  1;
                next_j      <=  0;
                -- outputs
                a_addr_o    <=  base_i + to_unsigned(i, address_bits);
                z_addr_o    <=  base_i + to_unsigned(j + 1, address_bits);
                z_o         <=  b_i;
                wr_o        <=  '0';
                sort_done_o <=  '0';
            when PREP_INNER =>
                -- internal signals
                next_tmp    <=  a_i;
                next_i      <=  i;
                next_j      <=  i - 1;
                -- outputs
                a_addr_o    <=  base_i + to_unsigned(i, address_bits);
                z_addr_o    <=  base_i + to_unsigned(j + 1, address_bits);
                z_o         <=  b_i;
                wr_o        <=  '0';
                sort_done_o <=  '0';
            when CHECK_INNER =>
                -- internal signals
                next_tmp    <=  tmp;
                next_i      <=  i;
                next_j      <=  j;
                -- outputs
                a_addr_o    <=  base_i + to_unsigned(i, address_bits);
                z_addr_o    <=  base_i + to_unsigned(j + 1, address_bits);
                z_o         <=  b_i;
                wr_o        <=  '0';
                sort_done_o <=  '0';
            when LOOP_INNER =>
                -- internal signals
                next_tmp    <=  tmp;
                next_i      <=  i;
                next_j      <=  j - 1;
                -- outputs
                a_addr_o    <=  base_i + to_unsigned(j - 1, address_bits);
                z_addr_o    <=  base_i + to_unsigned(j + 1, address_bits);
                z_o         <=  b_i;
                wr_o        <=  '1';
                sort_done_o <=  '0';
            when LAST_INNER =>
                -- internal signals
                next_tmp    <=  tmp;
                next_i      <=  i;
                next_j      <=  j;
                -- outputs
                a_addr_o    <=  base_i + to_unsigned(i, address_bits);
                z_addr_o    <=  base_i + to_unsigned(j + 1, address_bits);
```

```
                z_o          <=  b_i;
                wr_o         <=  '1';
                sort_done_o <=  '0';
            when SAVE_0 =>
                -- internal signals
                next_tmp     <=  tmp;
                next_i       <=  i + 1;
                next_j       <=  j;
                -- outputs
                a_addr_o     <=  base_i + to_unsigned(i, address_bits);
                z_addr_o     <=  base_i + to_unsigned(j, address_bits);
                z_o          <=  tmp;
                wr_o         <=  '1';
                sort_done_o <=  '0';
            when SAVE_1 =>
                -- internal signals
                next_tmp     <=  tmp;
                next_i       <=  i + 1;
                next_j       <=  j;
                --outputs
                a_addr_o     <=  base_i + to_unsigned(i, address_bits);
                z_addr_o     <=  base_i + to_unsigned(j + 1, address_bits);
                z_o          <=  tmp;
                wr_o         <=  '1';
                sort_done_o <=  '0';
            when DONE =>
                -- internal signals
                next_tmp     <=  tmp;
                next_i       <=  i;
                next_j       <=  j;
                -- outputs
                a_addr_o     <=  base_i + to_unsigned(i, address_bits);
                z_addr_o     <=  base_i + to_unsigned(j + 1, address_bits);
                z_o          <=  tmp;
                wr_o         <=  '0';
                sort_done_o <=  '1';
            when others =>  -- behave like IDLE
                -- internal signals
                next_tmp     <=  tmp;
                next_i       <=  1;
                next_j       <=  0;
                -- outputs
                a_addr_o     <=  base_i + to_unsigned(i, address_bits);
                z_addr_o     <=  base_i + to_unsigned(j + 1, address_bits);
                z_o          <=  b_i;
                wr_o         <=  '0';
                sort_done_o <=  '0';
        end case;
    end process output_comb;

    output_seq: process(rst_n_i, clk_i)
    begin
        if rst_n_i = '0' then
            tmp          <=  (others => '0');
            i            <=  0;
            j            <=  0;
        elsif rising_edge(clk_i) then
            tmp          <=  next_tmp;
            i            <=  next_i;
            j            <=  next_j;
        end if;
    end process output_seq;
```

```

end architecture rtl;
```

A.1.2 Circulant sum

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
library work;
    use work.system_params.all;
    use work.matrix_types.all;
    use work.matrix_mult_functions.all;

entity circulant_sum is
    port(
        -- control signals
        clk_i:      in   std_logic;
        rst_n_i:    in   std_logic;
        start_i:    in   std_logic;
        -- function arguments in (not latched)
        a_limit_i:  in   natural;     -- number of elements in 1st matrix
        a_base_i:   in   addr;        -- base address of 1st matrix
        b_limit_i:  in   natural;     -- number of elements in 2nd matrix
        b_base_i:   in   addr;        -- base address of 2nd matrix
        z_base_i:   in   addr;        -- base address of result matrix
        -- data, addresses and controls to memory
        a_i:        in   pos;
        a_addr_o:   out  addr;
        b_i:        in   pos;
        b_addr_o:   out  addr;
        z_o:        out  pos;
        z_addr_o:   out  addr;
        wr_o:       out  std_logic;   -- write enable for memory
        -- report back once done
        z_limit_o:  out  natural;     -- number of remaining elements (minus one)
        sum_done_o: out  std_logic
    );
end entity circulant_sum;

architecture rtl of circulant_sum is

    constant    INVALID_POS:       pos:=       to_unsigned(P, position_bits);

    signal  i:              natural;
    signal  j:              natural;
    signal  k:              natural;
    signal  next_i:         natural;
    signal  next_j:         natural;
    signal  next_k:         natural;

    signal  i_not_done:     std_logic;
    signal  j_not_done:     std_logic;
    signal  neither_done:   std_logic;

    type state_t is (IDLE, COMPARE, SKIP, COPY_I, COPY_J, FILL_INVALID, DONE);
    signal  state:          state_t;
    signal  next_state:     state_t;

begin
```

```vhdl

    i_not_done      <=  '1' when i < a_limit_i and a_i /= INVALID_POS else
                        '0';
    j_not_done      <=  '1' when j < b_limit_i and b_i /= INVALID_POS else
                        '0';
    neither_done    <=  i_not_done and j_not_done;

    a_addr_o    <=  a_base_i + i;
    b_addr_o    <=  b_base_i + j;
    z_addr_o    <=  z_base_i + k;

    z_limit_o   <=  k;

    state_comb: process(
        state, start_i, i_not_done, j_not_done, neither_done,
        k, a_i, b_i
    )
    begin
        case state is
            when IDLE =>
                if start_i = '1' then
                    next_state  <=  COMPARE;
                else
                    next_state  <=  IDLE;
                end if;
            when COMPARE =>
                if neither_done = '1' and a_i = b_i then
                    next_state  <=  SKIP;
                elsif neither_done = '1' and a_i < b_i then
                    next_state  <=  COPY_I;
                elsif neither_done = '1' and a_i > b_i then
                    next_state  <=  COPY_J;
                elsif i_not_done = '1' and not j_not_done = '1' then
                    next_state  <=  COPY_I;
                elsif j_not_done = '1' and i_not_done = '0' then
                    next_state  <=  COPY_J;
                elsif i_not_done = '0' and j_not_done = '0'  -- cont.
                        and k < sum(M)*DV then
                    next_state  <=  FILL_INVALID;
                else
                    next_state  <=  DONE;
                end if;
            when SKIP =>
                next_state  <=  COMPARE;
            when COPY_I =>
                next_state  <=  COMPARE;
            when COPY_J =>
                next_state  <=  COMPARE;
            when FILL_INVALID =>
                next_state  <=  COMPARE;
            when DONE =>
                if start_i = '0' then
                    next_state  <=  IDLE;
                else
                    next_state  <=  DONE;
                end if;
            when others =>
                next_state  <=  IDLE;
        end case;
    end process state_comb;

    state_seq: process(clk_i, rst_n_i)
```

```
    begin
        if rst_n_i = '0' then
            state    <=  IDLE;
        elsif rising_edge(clk_i) then
            state    <=   next_state;
        end if;
    end process state_seq;

    output_comb: process(state, i, j, k, a_i, b_i)
    begin
        case state is
            when IDLE =>
                -- internal signals
                next_i      <=  0;
                next_j      <=  0;
                next_k      <=  0;
                -- outputs
                z_o         <=  INVALID_POS;
                wr_o        <=  '0';
                sum_done_o  <=  '0';
            when COMPARE =>
                -- internal signals
                next_i      <=  i;
                next_j      <=  j;
                next_k      <=  k;
                -- outputs
                z_o         <=  INVALID_POS;
                wr_o        <=  '0';
                sum_done_o  <=  '0';
            when SKIP =>
                -- internal signals
                next_i      <=  i + 1;
                next_j      <=  j + 1;
                next_k      <=  k;
                -- outputs
                z_o         <=  INVALID_POS;
                wr_o        <=  '0';
                sum_done_o  <=  '0';
            when COPY_I =>
                -- internal signals
                next_i      <=  i + 1;
                next_j      <=  j;
                next_k      <=  k + 1;
                -- outputs
                z_o         <=  a_i;
                wr_o        <=  '1';
                sum_done_o  <=  '0';
            when COPY_J =>
                -- internal signals
                next_i      <=  i;
                next_j      <=  j + 1;
                next_k      <=  k + 1;
                -- outputs
                z_o         <=  b_i;
                wr_o        <=  '1';
                sum_done_o  <=  '0';
            when FILL_INVALID =>
                -- internal signals
                next_i      <=  i;
                next_j      <=  j;
                next_k      <=  k + 1;
                -- outputs
```

```
                    z_o         <=  INVALID_POS;
                    wr_o        <=  '1';
                    sum_done_o  <=  '0';
                when DONE =>
                    -- internal signals
                    next_i      <=  i;
                    next_j      <=  j;
                    next_k      <=  k + 1;
                    -- outputs
                    z_o         <=  INVALID_POS;
                    wr_o        <=  '0';
                    sum_done_o  <=  '1';
                when others =>
                    -- internal signals
                    next_i      <=  0;
                    next_j      <=  0;
                    next_k      <=  0;
                    -- outputs
                    z_o         <=  INVALID_POS;
                    wr_o        <=  '0';
                    sum_done_o  <=  '0';
            end case;
    end process output_comb;

    output_seq: process(clk_i, rst_n_i)
    begin
        if rst_n_i = '0' then
            i           <=  0;
            j           <=  0;
            k           <=  0;
        elsif rising_edge(clk_i) then
            i           <=  next_i;
            j           <=  next_j;
            k           <=  next_k;
        end if;
    end process output_seq;

end architecture rtl;
```

A.1.3 Memory copy

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
library work;
    use work.matrix_types.all;

entity mem_copy is
    port (
        -- control signals
        clk_i:          in  std_logic;
        rst_n_i:        in  std_logic;
        start_i:        in  std_logic;
        -- function arguments in (not latched)
        limit_i:        in  natural;    -- number of elements in source array
        a_base_i:       in  addr;       -- base addr of source array
        z_base_i:       in  addr;       -- base addr of destination array
        -- data, addresses and controls to memory
        a_i:            in  pos;        -- element A from source array
```

```vhdl
        a_addr_o:       out addr;       -- address of element A
        z_o:            out pos;        -- element Z to destination array
        z_addr_o:       out addr;       -- address of element Z
        wr_o:           out std_logic;  -- write enable for memory
        -- report back once done
        copy_done_o:    out std_logic   -- high when done
    );
end entity mem_copy;

architecture rtl of mem_copy is

    signal  i:          natural;
    signal  next_i:     natural;

    type state_t is (IDLE, COPY, DONE);
    signal  state:      state_t;
    signal  next_state: state_t;

begin

    a_addr_o    <=  a_base_i + i;
    z_addr_o    <=  z_base_i + i;
    z_o         <=  a_i;

    state_comb: process(state, start_i, i, limit_i)
    begin
        case state is
            when IDLE =>
                if start_i = '1' then
                    next_state  <=  COPY;
                else
                    next_state  <=  IDLE;
                end if;
            when COPY =>
                if i = limit_i-1 then
                    next_state  <=  DONE;
                else
                    next_state  <=  COPY;
                end if;
            when DONE =>
                if start_i = '0' then
                    next_state  <=  IDLE;
                else
                    next_state  <=  DONE;
                end if;
            when others =>
                next_state  <=  IDLE;
        end case;
    end process state_comb;

    state_seq: process(clk_i, rst_n_i)
    begin
        if rst_n_i = '0' then
            state   <=  IDLE;
        elsif rising_edge(clk_i) then
            state   <=  next_state;
        end if;
    end process state_seq;

    output_comb: process(state, i)
    begin
        case state is
```

```
            when IDLE =>
                -- internal signals
                next_i      <=  0;
                -- outputs
                wr_o        <=  '0';
                copy_done_o <=  '0';
            when COPY =>
                -- internal signals
                next_i      <=  i + 1;
                -- outputs
                wr_o        <=  '1';
                copy_done_o <=  '0';
            when DONE =>
                -- internal signals
                next_i      <=  0;
                -- outputs
                wr_o        <=  '0';
                copy_done_o <=  '1';
            when others =>
                -- internal signals
                next_i      <=  i;
                -- outputs
                wr_o        <=  '0';
                copy_done_o <=  '0';
        end case;
    end process output_comb;

    output_seq: process(clk_i, rst_n_i)
    begin
        if rst_n_i = '0' then
            i   <=  0;
        elsif rising_edge(clk_i) then
            i   <=  next_i;
        end if;
    end process output_seq;

end architecture rtl;
```

A.2 Vector by matrix

A.2.1 Vector by circulant

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
library work;
    use work.system_params.all;
    use work.matrix_types.all;

entity vector_by_circulant is
    port(
        -- control signals
        clk_i:          in  std_logic;
        rst_n_i:        in  std_logic;
        start_i:        in  std_logic;
        binary_i:       in  std_logic;
        -- function arguments in (not latched)
```

```vhdl
        a_base_i:       in   addr;
        b_base_i:       in   addr;
        b_weight:       in   natural;
        z_base_i:       in   addr;
        op:             in   std_logic;  -- '0' -> z=aB; '1' -> z=Ba
        -- data, addresses and controls to memory
        a_i:            in   pos;
        a_addr_o:       out  addr;
        b_i:            in   pos;
        b_addr_o:       out  addr;
        z_o:            out  pos;
        z_addr_o:       out  addr;
        wr_o:           out  std_logic;
        done_o:         out  std_logic
    );
end entity vector_by_circulant;

architecture arch of vector_by_circulant is

    type state_t is (IDLE, GET_M, INNER_LOOP, DONE);
    signal  state:      state_t;
    signal  next_state: state_t;

    signal  m_index_c, m_index_r:   natural;
    signal  m_c, m_r:   pos;
    signal  n_c, n_r:   addr;
    signal  i_c, i_r:   addr;

begin

    comb: process(state, start_i, binary_i,
        m_index_r, m_c, m_r, n_c, n_r, i_c, i_r,
        op, b_weight, a_base_i, b_base_i, z_base_i,
        a_i, b_i)
    begin
        next_state  <=  state;
        wr_o        <=  '0';
        done_o      <=  '0';
        m_index_c   <=  m_index_r;
        a_addr_o    <=  to_unsigned(0, address_bits);
        b_addr_o    <=  to_unsigned(0, address_bits);
        z_o         <=  to_unsigned(0, position_bits);
        z_addr_o    <=  to_unsigned(0, address_bits);
        m_c         <=  m_r;
        n_c         <=  n_r;
        i_c         <=  i_r;
        case state is
            when IDLE =>
                m_index_c   <=  0;
                m_c         <=  to_unsigned(0, position_bits);
                n_c         <=  to_unsigned(0, address_bits);
                i_c         <=  to_unsigned(0, address_bits);
                if start_i = '1' then
                    next_state  <=  GET_M;
                end if;
            when GET_M =>
                m_index_c   <=  m_index_r + 1;
                b_addr_o    <=  b_base_i + m_index_r;
                m_c         <=  b_i;
                n_c         <=  to_unsigned(0, address_bits);
                if op = '0' then
                    i_c     <=  to_unsigned(to_integer(m_c), address_bits);
```

```
                    else
                        i_c      <=  to_unsigned(to_integer(P-m_c-1), -- cont.
                                        address_bits);
                    end if;
                    if b_i = P then
                        next_state  <=  DONE;
                    else
                        next_state  <=  INNER_LOOP;
                    end if;
                when INNER_LOOP =>
                    wr_o     <=  '1';
                    n_c      <=  n_r + 1;
                    if i_r = P - 1 then
                        i_c <=  to_unsigned(0, address_bits);
                    else
                        i_c <=  i_r + 1;
                    end if;
                    a_addr_o    <=  a_base_i + n_c;
                    b_addr_o    <=  z_base_i + i_c;
                    z_addr_o    <=  z_base_i + i_c;
                    if binary_i = '0' then
                        z_o      <=  a_i + b_i;
                    else
                        z_o      <=  (0 => a_i(0) xor b_i(0), others => '0');
                    end if;
                    if n_r = P-1 then
                        if m_index_r = b_weight then
                            next_state  <=  DONE;
                        else
                            next_state  <=  GET_M;
                        end if;
                    end if;
                when DONE =>
                    done_o  <=  '1';
                    if start_i = '0' then
                        next_state  <=  IDLE;
                    end if;
                when others =>
                    next_state  <=  IDLE;
            end case;
    end process comb;

    seq: process(rst_n_i, clk_i)
    begin
        if rst_n_i = '0' then
            state       <=  IDLE;
            m_index_r   <=  0;
            m_r         <=  to_unsigned(0, position_bits);
            n_r         <=  to_unsigned(0, address_bits);
            i_r         <=  to_unsigned(0, address_bits);
        elsif rising_edge(clk_i) then
            state       <=  next_state;
            m_index_r   <=  m_index_c;
            m_r         <=  m_c;
            n_r         <=  n_c;
            i_r         <=  i_c;
        end if;
    end process seq;

end architecture;
```

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

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
library work;
    use work.system_params.all;
    use work.matrix_types.all;
    use work.matrix_mult_functions.all;
    use work.memory_map.all;

entity x_by_Ltr is
    port (
        -- control signals
        clk_i:          in  std_logic;
        rst_n_i:        in  std_logic;
        start_i:        in  std_logic;
        -- controls to multiplication core
        vector_base_o:  out addr;
        block_base_o:   out addr;
        result_base_o:  out addr;
        start_mul_o:    out std_logic;
        mul_done_i:     in  std_logic;
        -- report back
        done_o:         out std_logic
    );
end entity x_by_Ltr;

architecture rtl of x_by_Ltr is

    type state_t is (IDLE, BUSY, DONE);
    signal  state:      state_t;
    signal  next_state: state_t;

    signal  block_count_c, block_count_r:   natural;

    signal  vector_base_c, vector_base_r:   addr;
    signal  block_base_c, block_base_r:     addr;

begin

    vector_base_o   <=  vector_base_r;
    block_base_o    <=  block_base_r;
    result_base_o   <=  S_BASE_ADDR;

    comb: process(state, start_i,
        block_count_r, block_base_r, vector_base_r,
        mul_done_i)
    begin
        start_mul_o     <=  '0';
        next_state      <=  state;
        block_count_c   <=  block_count_r;
        block_base_c    <=  block_base_r;
        vector_base_c   <=  vector_base_r;
        done_o          <=  '0';
        case state is
            when IDLE =>
                block_count_c   <=  0;
                block_base_c    <=  LTR_BASE_ADDR;
                vector_base_c   <=  X_BASE_ADDR;
                if start_i = '1' then
```

```
                    next_state  <=  BUSY;
                end if;
            when BUSY =>
                start_mul_o <=  '1';
                if mul_done_i = '1' then
                    block_count_c   <=  block_count_r + 1;
                    -- M*DV is the maximum number of elements in a block of Ltr
                    block_base_c    <=  block_base_r + sum(M)*DV;
                    vector_base_c   <=  vector_base_r + P;
                    start_mul_o     <=  '0';
                    if block_count_r = N0-1 then
                        next_state  <=  DONE;
                    end if;
                end if;
            when DONE =>
                done_o  <=  '1';
                if start_i = '0' then
                    next_state  <=  IDLE;
                end if;
            when others =>
                next_state  <=  IDLE;
        end case;
    end process comb;

    seq: process(clk_i, rst_n_i)
    begin
        if rst_n_i = '0' then
            state           <=  IDLE;
            block_count_r   <=  0;
            block_base_r    <=  to_unsigned(0, address_bits);
            vector_base_r   <=  to_unsigned(0, address_bits);
        elsif rising_edge(clk_i) then
            state           <=  next_state;
            block_count_r   <=  block_count_c;
            block_base_r    <=  block_base_c;
            vector_base_r   <=  vector_base_c;
        end if;
    end process seq;

end architecture rtl;
```

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

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
library work;
    use work.system_params.all;
    use work.matrix_types.all;
    use work.matrix_mult_functions.all;
    use work.memory_map.all;

entity Htr_by_str is
    port (
        -- control signals
        clk_i:          in  std_logic;
        rst_n_i:        in  std_logic;
        start_i:        in  std_logic;
        -- controls to multiplication core
```

```
        vector_base_o:  out addr;
        block_base_o:   out addr;
        result_base_o:  out addr;
        start_mul_o:    out std_logic;
        mul_done_i:     in  std_logic;
        -- report back
        done_o:         out std_logic
    );
end entity Htr_by_str;

architecture rtl of Htr_by_str is

    type state_t is (IDLE, BUSY, DONE);
    signal  state:      state_t;
    signal  next_state: state_t;

    signal  block_count_c, block_count_r:   natural;

    signal  vector_base_c, vector_base_r:   addr;
    signal  block_base_c, block_base_r:     addr;
    signal  result_base_c, result_base_r:   addr;

begin

    vector_base_o   <=  vector_base_r;
    block_base_o    <=  block_base_r;
    result_base_o   <=  result_base_r;

    comb: process(state, start_i,
        block_count_r, block_base_r, vector_base_r, result_base_r,
        mul_done_i)
    begin
        start_mul_o     <=  '0';
        next_state      <=  state;
        block_count_c   <=  block_count_r;
        block_base_c    <=  block_base_r;
        vector_base_c   <=  vector_base_r;
        result_base_c   <=  result_base_r;
        done_o          <=  '0';
        case state is
            when IDLE =>
                block_count_c   <=  0;
                block_base_c    <=  HTR_BASE_ADDR;
                vector_base_c   <=  S_BASE_ADDR;
                result_base_c   <=  SIGMA_BASE_ADDR;
                if start_i = '1' then
                    next_state  <=  BUSY;
                end if;
            when BUSY =>
                start_mul_o <=  '1';
                if mul_done_i = '1' then
                    block_count_c   <=  block_count_r + 1;
                    block_base_c    <=  block_base_r + DV;
                    vector_base_c   <=  vector_base_r + P;
                    result_base_c   <=  result_base_r + P;
                    start_mul_o     <=  '0';
                    if block_count_r = N0-1 then
                        next_state  <=  DONE;
                    end if;
                end if;
            when DONE =>
                done_o  <=  '1';
```

```
                if start_i = '0' then
                    next_state  <=  IDLE;
                end if;
            when others =>
                next_state  <=  IDLE;
        end case;
    end process comb;

    seq: process(clk_i, rst_n_i)
    begin
        if rst_n_i = '0' then
            state           <=  IDLE;
            block_count_r   <=  0;
            block_base_r    <=  HTR_BASE_ADDR;
            vector_base_r   <=  S_BASE_ADDR;
            result_base_r   <=  SIGMA_BASE_ADDR;
        elsif rising_edge(clk_i) then
            state           <=  next_state;
            block_count_r   <=  block_count_c;
            block_base_r    <=  block_base_c;
            vector_base_r   <=  vector_base_c;
            result_base_r   <=  result_base_c;
        end if;
    end process seq;

end architecture rtl;
```

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

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
library work;
    use work.system_params.all;
    use work.matrix_types.all;
    use work.matrix_mult_functions.all;
    use work.memory_map.all;

entity Qtr_by_sigmatr is
    port (
        -- control signals
        clk_i:          in  std_logic;
        rst_n_i:        in  std_logic;
        start_i:        in  std_logic;
        -- controls to multiplication core
        vector_base_o:  out addr;
        block_base_o:   out addr;
        block_weight_o: out natural;
        result_base_o:  out addr;
        start_mul_o:    out std_logic;
        mul_done_i:     in  std_logic;
        -- report back
        done_o:         out std_logic
    );
end entity Qtr_by_sigmatr;

architecture rtl of Qtr_by_sigmatr is

    type state_t is (IDLE, BUSY, DONE);
```

```
    signal  state:       state_t;
    signal  next_state:  state_t;

    signal  block_row_c, block_row_r:        natural;
    signal  block_col_c, block_col_r:        natural;

    signal  Qtr_block_base_map:
        addr_matrix(N0-1 downto 0, N0-1 downto 0)   :=  get_Qtr_block_base;
    signal  Qtr_block_limit_map:
        n_matrix(N0-1 downto 0, N0-1 downto 0)      :=  get_block_limits;

    signal  vector_base_c, vector_base_r:   addr;
    signal  block_base_c, block_base_r:     addr;
    signal  result_base_c, result_base_r:   addr;

begin

    vector_base_o   <=  vector_base_r;
    block_base_o    <=  block_base_r;
    block_weight_o  <=  Qtr_block_limit_map(block_row_r, block_col_r);
    result_base_o   <=  result_base_r;

    comb: process(state, start_i,
        block_base_r, vector_base_r, result_base_r,
        block_row_r, block_col_r,
        mul_done_i)
    begin
        start_mul_o     <=  '0';
        next_state      <=  state;
        block_row_c     <=  block_row_r;
        block_col_c     <=  block_col_r;
        block_base_c    <=  block_base_r;
        vector_base_c   <=  vector_base_r;
        result_base_c   <=  result_base_r;
        done_o          <=  '0';
        case state is
            when IDLE =>
                block_row_c     <=  0;
                block_col_c     <=  0;
                block_base_c    <=  QTR_BASE_ADDR;
                vector_base_c   <=  SIGMA_BASE_ADDR;
                result_base_c   <=  R_BASE_ADDR;
                if start_i = '1' then
                    next_state  <=  BUSY;
                end if;
            when BUSY =>
                start_mul_o <=  '1';
                if mul_done_i = '1' then
                    start_mul_o     <=  '0';
                    if block_col_r /= N0-1 then
                        block_col_c     <=  block_col_r + 1;
                        block_base_c    <=  QTR_BASE_ADDR +  -- cont.
                            Qtr_block_base_map(block_row_r, block_col_r);
                        vector_base_c   <=  vector_base_r + P;
                        result_base_c   <=  result_base_r;
                    else
                        if block_row_r /= N0-1 then
                            block_row_c     <=  block_row_r + 1;
                            block_col_c     <=  0;
                            block_base_c    <=  QTR_BASE_ADDR + -- cont.
                                Qtr_block_base_map(block_row_r + 1, -- cont.
                                    block_col_r);
```

```
                            vector_base_c   <=  SIGMA_BASE_ADDR;
                            result_base_c   <=  result_base_r + P;
                        else
                            next_state  <=  DONE;
                        end if;
                    end if;
                end if;
            when DONE =>
                done_o  <=  '1';
                if start_i = '0' then
                    next_state  <=  IDLE;
                end if;
            when others =>
                next_state  <=  IDLE;
        end case;
    end process comb;

    seq: process(clk_i, rst_n_i)
    begin
        if rst_n_i = '0' then
            state           <=  IDLE;
            block_row_r     <=  0;
            block_col_r     <=  0;
            block_base_r    <=  HTR_BASE_ADDR;
            vector_base_r   <=  S_BASE_ADDR;
            result_base_r   <=  SIGMA_BASE_ADDR;
        elsif rising_edge(clk_i) then
            state           <=  next_state;
            block_base_r    <=  block_base_c;
            vector_base_r   <=  vector_base_c;
            result_base_r   <=  result_base_c;
            block_row_r     <=  block_row_c;
            block_col_r     <=  block_col_c;
        end if;
    end process seq;

end architecture rtl;
```

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

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
library work;
    use work.system_params.all;
    use work.matrix_types.all;
    use work.matrix_mult_functions.all;
    use work.memory_map.all;

entity e_by_Htr is
    port (
        -- control signals
        clk_i:          in  std_logic;
        rst_n_i:        in  std_logic;
        start_i:        in  std_logic;
        -- controls to multiplication core
        vector_base_o:  out addr;
        block_base_o:   out addr;
        result_base_o:  out addr;
```

```
        start_mul_o:     out std_logic;
        mul_done_i:      in  std_logic;
        -- report back
        done_o:          out std_logic
    );
end entity e_by_Htr;

architecture rtl of e_by_Htr is

    type state_t is (IDLE, BUSY, DONE);
    signal state:       state_t;
    signal next_state: state_t;

    signal block_count_c, block_count_r:   natural;

    signal vector_base_c, vector_base_r:   addr;
    signal block_base_c, block_base_r:     addr;

begin

    vector_base_o   <=  vector_base_r;
    block_base_o    <=  block_base_r;
    result_base_o   <=  S_BASE_ADDR;

    comb: process(state, start_i,
        block_count_r, block_base_r, vector_base_r,
        mul_done_i)
    begin
        start_mul_o     <=  '0';
        next_state      <=  state;
        block_count_c   <=  block_count_r;
        block_base_c    <=  block_base_r;
        vector_base_c   <=  vector_base_r;
        done_o          <=  '0';
        case state is
            when IDLE =>
                block_count_c   <=  0;
                block_base_c    <=  HTR_BASE_ADDR;
                vector_base_c   <=  E_BASE_ADDR;
                if start_i = '1' then
                    next_state  <=  BUSY;
                end if;
            when BUSY =>
                start_mul_o <=  '1';
                if mul_done_i = '1' then
                    block_count_c   <=  block_count_r + 1;
                    block_base_c    <=  block_base_r + DV;
                    vector_base_c   <=  vector_base_r + P;
                    start_mul_o     <=  '0';
                    if block_count_r = N0-1 then
                        next_state  <=  DONE;
                    end if;
                end if;
            when DONE =>
                done_o  <=  '1';
                if start_i = '0' then
                    next_state  <=  IDLE;
                end if;
            when others =>
                next_state  <=  IDLE;
        end case;
    end process comb;
```

```

    seq: process(clk_i, rst_n_i)
    begin
        if rst_n_i = '0' then
            state           <=  IDLE;
            block_count_r   <=  0;
            block_base_r    <=  to_unsigned(0, address_bits);
            vector_base_r   <=  to_unsigned(0, address_bits);
        elsif rising_edge(clk_i) then
            state           <=  next_state;
            block_count_r   <=  block_count_c;
            block_base_r    <=  block_base_c;
            vector_base_r   <=  vector_base_c;
        end if;
    end process seq;

end architecture rtl;
```

A.3 Error update

Peak search

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
library work;
    use work.system_params.all;
    use work.matrix_types.all;
    use work.matrix_mult_functions.all;
    use work.memory_map.all;

entity find_max is
    port(
        -- control signals
        clk_i:      in  std_logic;
        rst_n_i:    in  std_logic;
        start_i:    in  std_logic;
        -- to memory
        a_i:        in  pos;
        a_addr_o:   out addr;
        b_i:        in  pos;
        b_addr_o:   out addr;
        z_o:        out pos;
        z_addr_o:   out addr;
        wr_o:       out std_logic;
        -- report back when done
        max_idx_o:  out n_array(15 downto 0);
        done_o:     out std_logic
    );
end entity find_max;

architecture rtl of find_max is

    type state_t is (IDLE, BUSY, DONE);
    signal  state:      state_t;
    signal  next_state: state_t;

```

```
    signal  a_index_c, a_index_r:   natural;

    signal  max_val_c, max_val_r:           pos;
    signal  max_indexes_c, max_indexes_r:   n_array(15 downto 0);

    signal  i_c, i_r:   natural;

    constant    INVALID:    natural:=   N0*P;

begin

    a_addr_o    <=  R_BASE_ADDR + a_index_r;
    b_addr_o    <=  to_unsigned(0, address_bits);
    z_o         <=  to_unsigned(0, position_bits);
    z_addr_o    <=  to_unsigned(0, address_bits);
    wr_o        <=  '0';
    max_idx_o   <=  max_indexes_r;

    seq: process(clk_i, rst_n_i)
    begin
        if rst_n_i = '0' then
            state           <=  IDLE;
            a_index_r       <=  0;
            max_val_r       <=  to_unsigned(0, position_bits);
            max_indexes_r   <=  (others => invalid);
            i_r             <=  0;
        elsif rising_edge(clk_i) then
            state           <=  next_state;
            a_index_r       <=  a_index_c;
            max_val_r       <=  max_val_c;
            max_indexes_r   <=  max_indexes_c;
            i_r             <=  i_c;
        end if;
    end process seq;
    comb: process(
        state, start_i,
        a_index_r, a_i,
        max_val_r, max_indexes_r, i_r)
    begin
        next_state      <=  state;
        done_o          <=  '0';
        a_index_c       <=  0;
        max_val_c       <=  max_val_r;
        max_indexes_c   <=  max_indexes_r;
        i_c             <=  i_r;
        case state is
            when IDLE =>
                max_val_c       <=  to_unsigned(0, position_bits);
                max_indexes_c   <=  (others => INVALID);
                i_c             <=  0;
                if start_i = '1' then
                    next_state  <=  BUSY;
                end if;
            when BUSY =>
                a_index_c   <=  a_index_r + 1;
                if a_i = max_val_r then
                    max_indexes_c(i_r)  <=  a_index_r;
                    -- if we have too many equal maxes we'll flip just some
                    if i_r /= 15 then
                        i_c                 <=  i_r + 1;
                    end if;
                elsif a_i > max_val_r then
```

```
                    max_val_c       <=  a_i;
                    max_indexes_c   <=  (0 => a_index_r, others => INVALID);
                    i_c             <=  1;
                end if;
                if a_index_r = N0*P-1 then
                    a_index_c   <=  0;
                    next_state  <=  DONE;
                end if;
            when DONE =>
                done_o  <=  '1';
                if start_i = '0' then
                    next_state  <=  IDLE;
                end if;
        end case;
    end process comb;

end architecture rtl;
```

Vector plus rows

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
library work;
    use work.system_params.all;
    use work.matrix_types.all;
    use work.memory_map.all;
    use work.matrix_mult_functions.all;

entity vector_plus_rows is
    port (
        -- control signals
        clk_i:      in  std_logic;
        rst_n_i:    in  std_logic;
        start_i:    in  std_logic;
        -- row indexes
        row_idx_i:  in  n_array(15 downto 0);
        -- to memory
        a_i:        in  pos;
        a_addr_o:   out addr;
        b_i:        in  pos;
        b_addr_o:   out addr;
        z_o:        out pos;
        z_addr_o:   out addr;
        wr_o:       out std_logic;
        -- report back when done
        done_o:     out std_logic
    );
end entity vector_plus_rows;

architecture rtl of vector_plus_rows is

    type state_t is (IDLE, RELATIVE_ROW, SUM_ROW, DONE);
    signal  state:      state_t;
    signal  next_state: state_t;

    signal  cur_idx_c:  natural;
    signal  cur_idx_r:  natural;

```

```
    signal  block_row_c:    natural;
    signal  block_row_r:    natural;
    signal  block_col_c:    natural;
    signal  block_col_r:    natural;
    signal  rel_row_c:      natural;
    signal  rel_row_r:      natural;

    signal  i_c:    natural;
    signal  i_r:    natural;

    signal  Qtr_block_base:     addr;
    signal  Qtr_block_count:    natural;

    constant    Qtr_block_limit_map:    n_matrix:=      get_block_limits;
    constant    Qtr_block_base_map:     addr_matrix:=   get_Qtr_block_base;

    constant    INVALID:    natural:=   N0*P;

begin

    Qtr_block_count <=  Qtr_block_limit_map(block_row_r, block_col_r);
    Qtr_block_base  <=  Qtr_block_base_map(block_row_r, block_col_r);

    a_addr_o    <=  E_BASE_ADDR + block_col_r*P + ((b_i + rel_row_r) mod P);
    b_addr_o    <=  QTR_BASE_ADDR + Qtr_block_base + i_r;
    z_o         <=  (a_i + 1) mod 2;
    z_addr_o    <=  E_BASE_ADDR + block_col_r*P + ((b_i + rel_row_r) mod P);

    seq: process(clk_i, rst_n_i)
    begin
        if rst_n_i = '0' then
            state       <=  IDLE;
            cur_idx_r   <=  0;
            block_row_r <=  0;
            block_col_r <=  0;
            rel_row_r   <=  0;
            i_r         <=  0;
        elsif rising_edge(clk_i) then
            state       <=  next_state;
            cur_idx_r   <=  cur_idx_c;
            block_row_r <=  block_row_c;
            block_col_r <=  block_col_c;
            rel_row_r   <=  rel_row_c;
            i_r         <=  i_c;
        end if;
    end process seq;
    comb: process(
        state, start_i,
        block_row_r, block_col_r, rel_row_r,
        row_idx_i, cur_idx_r, i_r,
        Qtr_block_count)
    begin
        next_state  <=  state;
        cur_idx_c   <=  cur_idx_r;
        block_row_c <=  block_row_r;
        block_col_c <=  block_col_r;
        rel_row_c   <=  rel_row_r;
        i_c         <=  0;
        wr_o        <=  '0';
        done_o      <=  '0';
        case (state) is
            when IDLE =>
```

```
                    cur_idx_c    <=  0;
                    block_row_c <=  0;
                    block_col_c <=  0;
                    if start_i = '1' then
                        next_state  <=  RELATIVE_ROW;
                    end if;
                when RELATIVE_ROW =>
                    if row_idx_i(cur_idx_r) /= INVALID then
                        block_row_c <=  row_idx_i(cur_idx_r) / P;
                        rel_row_c   <=  row_idx_i(cur_idx_r) mod P;
                        next_state  <=  SUM_ROW;
                    else
                        next_state  <=  DONE;
                    end if;
                when SUM_ROW =>
                    wr_o        <=  '1';
                    if i_r /= Qtr_block_count then
                        i_c         <=  i_r + 1;
                    else
                        if block_col_r /= N0-1 then
                            block_col_c <=  block_col_r + 1;
                        else
                            if cur_idx_r = 15 then
                                next_state  <=  DONE;
                            else
                                cur_idx_c   <=  cur_idx_r + 1;
                                block_col_c <=  0;
                                next_state  <=  RELATIVE_ROW;
                            end if;
                        end if;
                    end if;
                when DONE =>
                    done_o      <=  '1';
                    if start_i = '0' then
                        next_state  <=  IDLE;
                    end if;
            end case;
        end process comb;

end architecture rtl;
```

A.4 Loop control and message computation

Zero syndrome detection

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
library work;
    use work.system_params.all;
    use work.matrix_types.all;
    use work.matrix_mult_functions.all;
    use work.memory_map.all;

entity null_syndrome is
    port(
        -- control signals
```

```vhdl
        clk_i:      in  std_logic;
        rst_n_i:    in  std_logic;
        start_i:    in  std_logic;
        -- to memory
        a_i:        in  pos;
        a_addr_o:   out addr;
        b_i:        in  pos;
        b_addr_o:   out addr;
        z_o:        out pos;
        z_addr_o:   out addr;
        wr_o:       out std_logic;
        -- report back when done
        null_syn_o: out std_logic;
        done_o:     out std_logic
    );
end entity null_syndrome;

architecture rtl of null_syndrome is

    type state_t is (IDLE, BUSY, DONE);
    signal  state:      state_t;
    signal  next_state: state_t;

    signal  bad_syn_c:  std_logic;
    signal  bad_syn_r:  std_logic;

    signal  i_c:    natural;
    signal  i_r:    natural;

begin

    a_addr_o    <=  S_BASE_ADDR + i_r;
    b_addr_o    <=  S_BASE_ADDR + i_r + 1;
    z_o         <=  to_unsigned(0, position_bits);
    z_addr_o    <=  S_BASE_ADDR;
    wr_o        <=  '0';
    null_syn_o  <=  not bad_syn_r;

    seq: process(clk_i, rst_n_i)
    begin
        if rst_n_i = '0' then
            state       <=  IDLE;
            bad_syn_r   <=  '0';
            i_r         <=  0;
        elsif rising_edge(clk_i) then
            state       <=  next_state;
            bad_syn_r   <=  bad_syn_c;
            i_r         <=  i_c;
        end if;
    end process seq;
    comb: process(state, start_i, a_i, b_i, i_r, bad_syn_c, bad_syn_r)
    begin
        next_state  <=  state;
        bad_syn_c   <=  bad_syn_r;
        i_c         <=  i_r;
        done_o      <=  '0';
        case state is
            when IDLE =>
                bad_syn_c   <=  '0';
                i_c         <=  0;
                if start_i = '1' then
                    next_state  <=  BUSY;
```

```
                end if;
            when BUSY =>
                i_c      <=  i_r + 2;
                if a_i /= 0 then
                    bad_syn_c   <=  '1';
                end if;
                if b_i /= 0 and i_r /= P-1 then
                    bad_syn_c   <=  '1';
                end if;
                if bad_syn_c = '1' or i_r = P-1 then
                    next_state  <=  DONE;
                end if;
            when DONE =>
                done_o      <=  '1';
                if start_i = '0' then
                    next_state  <=  IDLE;
                end if;
        end case;
    end process comb;

end architecture rtl;
```

Intermediate result clear

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
library work;
    use work.system_params.all;
    use work.matrix_types.all;
    use work.matrix_mult_functions.all;
    use work.memory_map.all;

entity clear_temp is
    port(
        -- control signals
        clk_i:      in  std_logic;
        rst_n_i:    in  std_logic;
        start_i:    in  std_logic;
        -- to memory
        a_i:        in  pos;
        a_addr_o:   out addr;
        b_i:        in  pos;
        b_addr_o:   out addr;
        z_o:        out pos;
        z_addr_o:   out addr;
        wr_o:       out std_logic;
        -- report back when done
        done_o:     out std_logic
    );
end entity clear_temp;

architecture rtl of clear_temp is

    type state_t is (
        IDLE, CLEAR_MUL_RES, CLEAR_SUM_TMP, CLEAR_SIGMA,
        CLEAR_R, DONE
    );
    signal  state:      state_t;
```

```
    signal  next_state: state_t;

    signal  i_c:        integer;
    signal  i_r:        integer;

begin

    a_addr_o    <=  HTR_BASE_ADDR;
    b_addr_o    <=  HTR_BASE_ADDR;
    z_o         <=  to_unsigned(0, position_bits);

    seq: process(clk_i, rst_n_i)
    begin
        if rst_n_i = '0' then
            state   <=  IDLE;
            i_r     <=  0;
        elsif rising_edge(clk_i) then
            state   <=  next_state;
            i_r     <=  i_c;
        end if;
    end process seq;
    comb: process(state, start_i, i_r)
    begin
        i_c     <=  i_r;
        done_o  <=  '0';
        case state is
            when IDLE =>
                i_c         <=  0;
                z_addr_o    <=  HTR_BASE_ADDR;
                wr_o        <=  '0';
                if start_i = '1' then
                    next_state  <=  CLEAR_MUL_RES;
                end if;
            when CLEAR_MUL_RES =>
                i_c         <=  i_r + 1;
                z_addr_o    <=  MUL_RES_BASE_ADDR + i_r;
                wr_o        <=  '1';
                if i_r = DV*max(M) - 1 then
                    i_c         <=  0;
                    next_state  <=  CLEAR_SUM_TMP;
                end if;
            when CLEAR_SUM_TMP =>
                i_c         <=  i_r + 1;
                z_addr_o    <=  SUM_TMP_BASE_ADDR + i_r;
                wr_o        <=  '1';
                if i_r = DV*sum(M) - 1 then
                    i_c         <=  0;
                    next_state  <=  CLEAR_SIGMA;
                end if;
            when CLEAR_SIGMA =>
                i_c         <=  i_r + 1;
                z_addr_o    <=  SIGMA_BASE_ADDR + i_r;
                wr_o        <=  '1';
                if i_r = N0*P - 1 then
                    i_c         <=  0;
                    next_state  <=  CLEAR_R;
                end if;
            when CLEAR_R =>
                i_c         <=  i_r + 1;
                z_addr_o    <=  MUL_RES_BASE_ADDR + i_r;
                wr_o        <=  '1';
                if i_r = N0*P - 1 then
```

```
                    i_c          <=  0;
                    next_state   <=  DONE;
                end if;
            when DONE =>
                i_c          <=  0;
                z_addr_o     <=  HTR_BASE_ADDR;
                wr_o         <=  '0';
                done_o       <=  '1';
                if start_i = '0' then
                    next_state   <=  IDLE;
                end if;
        end case;
    end process comb;

end architecture rtl;
```

Message computation

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
library work;
    use work.system_params.all;
    use work.matrix_types.all;
    use work.matrix_mult_functions.all;
    use work.memory_map.all;

entity comp_message is
    port(
        -- control signals
        clk_i:      in  std_logic;
        rst_n_i:    in  std_logic;
        start_i:    in  std_logic;
        -- to memory
        a_i:        in  pos;
        a_addr_o:   out addr;
        b_i:        in  pos;
        b_addr_o:   out addr;
        z_o:        out pos;
        z_addr_o:   out addr;
        wr_o:       out std_logic;
        -- report back when done
        done_o:     out std_logic
    );
end entity comp_message;

architecture rtl of comp_message is

    type state_t is (IDLE, BUSY, DONE);
    signal  state:      state_t;
    signal  next_state: state_t;

    signal  i_c, i_r:   integer;

begin

    a_addr_o    <=  X_BASE_ADDR + i_r;
    b_addr_o    <=  E_BASE_ADDR + i_r;
    z_o         <=  a_i xor b_i;
```

```
    z_addr_o    <=  U_BASE_ADDR + i_r;

    seq: process(clk_i, rst_n_i)
    begin
        if rst_n_i = '0' then
            state   <=  IDLE;
        elsif rising_edge(clk_i) then
            state   <=  next_state;
        end if;
    end process seq;
    comb: process(state, start_i, i_r)
    begin
        next_state  <=  state;
        i_c         <=  0;
        wr_o        <=  '0';
        done_o      <=  '0';
        case (state) is
            when IDLE =>
                if start_i = '1' then
                    next_state  <=  BUSY;
                end if;
            when BUSY =>
                i_c     <=  i_r + 1;
                wr_o    <=  '1';
                if i_r = N0*(P-1) - 1 then
                    next_state  <=  DONE;
                end if;
            when DONE =>
                done_o  <=  '1';
                if start_i = '0' then
                    next_state  <=  IDLE;
                end if;
        end case;
    end process comb;

end architecture rtl;
```

A.5 Top module

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.numeric_std.all;
library work;
    use work.system_params.all;
    use work.matrix_types.all;
    use work.matrix_mult_functions.all;

entity top is
    port(
        -- control signals
        clk_i:      in  std_logic;
        rst_n_i:    in  std_logic;
        start_i:    in  std_logic;
        -- to memory
        a_i:        in  pos;
        a_addr_o:   out addr;
        b_i:        in  pos;
        b_addr_o:   out addr;
```

```
        z_o:        out pos;
        z_addr_o:   out addr;
        wr_o:       out std_logic;
        -- report back when done
        failure_o:  out std_logic;
        done_o:     out std_logic
    );
end entity top;

architecture rtl of top is

    component key_reconstruction is
        port (
            -- control signals
            clk_i:          in   std_logic;
            rst_n_i:        in   std_logic;
            start_i:        in   std_logic;
            -- data, addresses and controls to memory
            a_i:            in   pos;
            a_addr_o:       out addr;
            b_i:            in   pos;
            b_addr_o:       out addr;
            z_o:            out pos;
            z_addr_o:       out addr;
            wr_o:           out std_logic;
            -- report back once done
            key_rec_done_o: out std_logic
        );
    end component key_reconstruction;
    signal  kr_start:   std_logic;
    signal  kr_a_addr:  addr;
    signal  kr_b_addr:  addr;
    signal  kr_z:       pos;
    signal  kr_z_addr:  addr;
    signal  kr_wr:      std_logic;
    signal  kr_done:    std_logic;

    component vector_by_circulant is
        port(
            -- control signals
            clk_i:          in   std_logic;
            rst_n_i:        in   std_logic;
            start_i:        in   std_logic;
            binary_i:       in   std_logic;
            -- function arguments in (not latched)
            a_base_i:       in   addr;
            b_base_i:       in   addr;
            b_weight:       in   integer;
            z_base_i:       in   addr;
            op:             in   std_logic;  -- '0' -> z=aB; '1' -> z=Ba
            -- data, addresses and controls to memory
            a_i:            in   pos;
            a_addr_o:       out addr;
            b_i:            in   pos;
            b_addr_o:       out addr;
            z_o:            out pos;
            z_addr_o:       out addr;
            wr_o:           out std_logic;
            done_o:         out std_logic
        );
    end component vector_by_circulant;
    signal  vc_start:       std_logic;
```

```
    signal  vc_binary:      std_logic;
    signal  vc_vec_base:    addr;
    signal  vc_blk_base:    addr;
    signal  vc_blk_weight:  natural;
    signal  vc_res_base:    addr;
    signal  vc_op:          std_logic;
    signal  vc_a_addr:      addr;
    signal  vc_b_addr:      addr;
    signal  vc_z:           pos;
    signal  vc_z_addr:      addr;
    signal  vc_wr:          std_logic;
    signal  vc_done:        std_logic;

    component x_by_Ltr is
        port (
            -- control signals
            clk_i:          in  std_logic;
            rst_n_i:        in  std_logic;
            start_i:        in  std_logic;
            -- controls to multiplication core
            vector_base_o:  out addr;
            block_base_o:   out addr;
            result_base_o:  out addr;
            start_mul_o:    out std_logic;
            mul_done_i:     in  std_logic;
            -- report back
            done_o:         out std_logic
        );
    end component x_by_Ltr;
    signal  xl_start:       std_logic;
    signal  xl_vec_base:    addr;
    signal  xl_blk_base:    addr;
    signal  xl_res_base:    addr;
    signal  xl_start_mul:   std_logic;
    signal  xl_done:        std_logic;

    component Htr_by_str is
        port (
            -- control signals
            clk_i:          in  std_logic;
            rst_n_i:        in  std_logic;
            start_i:        in  std_logic;
            -- controls to multiplication core
            vector_base_o:  out addr;
            block_base_o:   out addr;
            result_base_o:  out addr;
            start_mul_o:    out std_logic;
            mul_done_i:     in  std_logic;
            -- report back
            done_o:         out std_logic
        );
    end component Htr_by_str;
    signal  hs_start:       std_logic;
    signal  hs_vec_base:    addr;
    signal  hs_blk_base:    addr;
    signal  hs_res_base:    addr;
    signal  hs_start_mul:   std_logic;
    signal  hs_done:        std_logic;

    component Qtr_by_sigmatr is
        port (
            -- control signals
```

```
            clk_i:          in   std_logic;
            rst_n_i:        in   std_logic;
            start_i:        in   std_logic;
            -- controls to multiplication core
            vector_base_o:  out  addr;
            block_base_o:   out  addr;
            block_weight_o: out  natural;
            result_base_o:  out  addr;
            start_mul_o:    out  std_logic;
            mul_done_i:     in   std_logic;
            -- report back
            done_o:         out  std_logic
        );
    end component Qtr_by_sigmatr;
    signal  qs_start:       std_logic;
    signal  qs_vec_base:    addr;
    signal  qs_blk_base:    addr;
    signal  qs_blk_weight:  natural;
    signal  qs_res_base:    addr;
    signal  qs_start_mul:   std_logic;
    signal  qs_done:        std_logic;

    component find_max is
        port(
            -- control signals
            clk_i:      in   std_logic;
            rst_n_i:    in   std_logic;
            start_i:    in   std_logic;
            -- to memory
            a_i:        in   pos;
            a_addr_o:   out  addr;
            b_i:        in   pos;
            b_addr_o:   out  addr;
            z_o:        out  pos;
            z_addr_o:   out  addr;
            wr_o:       out  std_logic;
            -- report back when done
            max_idx_o:  out  n_array(15 downto 0);
            done_o:     out  std_logic
        );
    end component find_max;
    signal  fm_start:       std_logic;
    signal  fm_a_addr:      addr;
    signal  fm_b_addr:      addr;
    signal  fm_z:           pos;
    signal  fm_z_addr:      addr;
    signal  fm_wr:          std_logic;
    signal  fm_max_idx:     n_array(15 downto 0);
    signal  fm_done:        std_logic;
    signal  max_idx_c:      n_array(15 downto 0);
    signal  max_idx_r:      n_array(15 downto 0);

    component vector_plus_rows is
        port (
            -- control signals
            clk_i:      in   std_logic;
            rst_n_i:    in   std_logic;
            start_i:    in   std_logic;
            -- row indexes
            row_idx_i:  in   n_array(15 downto 0);
            -- to memory
            a_i:        in   pos;
```

```
            a_addr_o:   out addr;
            b_i:        in  pos;
            b_addr_o:   out addr;
            z_o:        out pos;
            z_addr_o:   out addr;
            wr_o:       out std_logic;
            -- report back when done
            done_o:     out std_logic
        );
    end component vector_plus_rows;
    signal  vr_start:       std_logic;
    signal  vr_a_addr:      addr;
    signal  vr_b_addr:      addr;
    signal  vr_z:           pos;
    signal  vr_z_addr:      addr;
    signal  vr_wr:          std_logic;
    signal  vr_done:        std_logic;

    component e_by_Htr is
        port (
            -- control signals
            clk_i:          in  std_logic;
            rst_n_i:        in  std_logic;
            start_i:        in  std_logic;
            -- controls to multiplication core
            vector_base_o:  out addr;
            block_base_o:   out addr;
            result_base_o:  out addr;
            start_mul_o:    out std_logic;
            mul_done_i:     in  std_logic;
            -- report back
            done_o:         out std_logic
        );
    end component e_by_Htr;
    signal  eh_start:       std_logic;
    signal  eh_vec_base:    addr;
    signal  eh_blk_base:    addr;
    signal  eh_res_base:    addr;
    signal  eh_start_mul:   std_logic;
    signal  eh_done:        std_logic;

    component null_syndrome is
        port(
            -- control signals
            clk_i:      in  std_logic;
            rst_n_i:    in  std_logic;
            start_i:    in  std_logic;
            -- to memory
            a_i:        in  pos;
            a_addr_o:   out addr;
            b_i:        in  pos;
            b_addr_o:   out addr;
            z_o:        out pos;
            z_addr_o:   out addr;
            wr_o:       out std_logic;
            -- report back when done
            null_syn_o: out std_logic;
            done_o:     out std_logic
        );
    end component null_syndrome;
    signal  ns_start:       std_logic;
    signal  ns_a_addr:      addr;
```

```
    signal  ns_b_addr:      addr;
    signal  ns_z:           pos;
    signal  ns_z_addr:      addr;
    signal  ns_wr:          std_logic;
    signal  ns_null_syn:    std_logic;
    signal  ns_done:        std_logic;

    component clear_temp is
        port(
            -- control signals
            clk_i:      in  std_logic;
            rst_n_i:    in  std_logic;
            start_i:    in  std_logic;
            -- to memory
            a_i:        in  pos;
            a_addr_o:   out addr;
            b_i:        in  pos;
            b_addr_o:   out addr;
            z_o:        out pos;
            z_addr_o:   out addr;
            wr_o:       out std_logic;
            -- report back when done
            done_o:     out std_logic
        );
    end component clear_temp;
    signal  ct_start:       std_logic;
    signal  ct_a_addr:      addr;
    signal  ct_b_addr:      addr;
    signal  ct_z:           pos;
    signal  ct_z_addr:      addr;
    signal  ct_wr:          std_logic;
    signal  ct_null_syn:    std_logic;
    signal  ct_done:        std_logic;

    component comp_message is
        port(
            -- control signals
            clk_i:      in  std_logic;
            rst_n_i:    in  std_logic;
            start_i:    in  std_logic;
            -- to memory
            a_i:        in  pos;
            a_addr_o:   out addr;
            b_i:        in  pos;
            b_addr_o:   out addr;
            z_o:        out pos;
            z_addr_o:   out addr;
            wr_o:       out std_logic;
            -- report back when done
            done_o:     out std_logic
        );
    end component comp_message;
    signal  cm_start:       std_logic;
    signal  cm_a_addr:      addr;
    signal  cm_b_addr:      addr;
    signal  cm_z:           pos;
    signal  cm_z_addr:      addr;
    signal  cm_wr:          std_logic;
    signal  cm_done:        std_logic;

    type state_t is (
        IDLE,
```

```
        COMPUTE_LTR,
        COMPUTE_INITIAL_SYNDROME,
        COMPUTE_SIGMA,
        COMPUTE_R,
        FIND_B,
        COMPUTE_ERROR,
        COMPUTE_SYNDROME,
        CHECK_SYNDROME,
        CLEAR_TEMP_AND_LOOP,
        COMPUTE_MESSAGE,
        CLEAR_TEMP_AND_RETURN,
        DONE
    );
    signal  state:      state_t;
    signal  next_state: state_t;

    signal  l_c:    integer;
    signal  l_r:    integer;

begin

    KR: key_reconstruction
        port map (
            clk_i               =>  clk_i,
            rst_n_i             =>  rst_n_i,
            start_i             =>  kr_start,
            a_i                 =>  a_i,
            a_addr_o            =>  kr_a_addr,
            b_i                 =>  b_i,
            b_addr_o            =>  kr_b_addr,
            z_o                 =>  kr_z,
            z_addr_o            =>  kr_z_addr,
            wr_o                =>  kr_wr,
            key_rec_done_o      =>  kr_done
        );

    VC: vector_by_circulant
        port map (
            clk_i           =>  clk_i,
            rst_n_i         =>  rst_n_i,
            start_i         =>  vc_start,
            binary_i        =>  vc_binary,
            a_base_i        =>  vc_vec_base,
            b_base_i        =>  vc_blk_base,
            b_weight        =>  vc_blk_weight,
            z_base_i        =>  vc_res_base,
            op              =>  vc_op,
            a_i             =>  a_i,
            a_addr_o        =>  vc_a_addr,
            b_i             =>  b_i,
            b_addr_o        =>  vc_b_addr,
            z_o             =>  vc_z,
            z_addr_o        =>  vc_z_addr,
            wr_o            =>  vc_wr,
            done_o          =>  vc_done
        );

    XL: x_by_Ltr
        port map (
            clk_i               =>  clk_i,
            rst_n_i             =>  rst_n_i,
            start_i             =>  xl_start,
```

```
            vector_base_o    =>  xl_vec_base,
            block_base_o     =>  xl_blk_base,
            result_base_o    =>  xl_res_base,
            start_mul_o      =>  xl_start_mul,
            mul_done_i       =>  vc_done,
            done_o           =>  xl_done
        );

    HS: Htr_by_str
        port map (
            clk_i            =>  clk_i,
            rst_n_i          =>  rst_n_i,
            start_i          =>  hs_start,
            vector_base_o    =>  hs_vec_base,
            block_base_o     =>  hs_blk_base,
            result_base_o    =>  hs_res_base,
            start_mul_o      =>  hs_start_mul,
            mul_done_i       =>  vc_done,
            done_o           =>  hs_done
        );

    QS: Qtr_by_sigmatr
        port map (
            clk_i            =>  clk_i,
            rst_n_i          =>  rst_n_i,
            start_i          =>  qs_start,
            vector_base_o    =>  qs_vec_base,
            block_base_o     =>  qs_blk_base,
            block_weight_o   =>  qs_blk_weight,
            result_base_o    =>  qs_res_base,
            start_mul_o      =>  qs_start_mul,
            mul_done_i       =>  vc_done,
            done_o           =>  qs_done
        );

    FM: find_max
        port map (
            clk_i            =>  clk_i,
            rst_n_i          =>  rst_n_i,
            start_i          =>  fm_start,
            a_i              =>  a_i,
            a_addr_o         =>  fm_a_addr,
            b_i              =>  b_i,
            b_addr_o         =>  fm_b_addr,
            z_o              =>  fm_z,
            z_addr_o         =>  fm_z_addr,
            wr_o             =>  fm_wr,
            max_idx_o        =>  fm_max_idx,
            done_o           =>  fm_done
        );

    VR: vector_plus_rows
        port map (
            clk_i            =>  clk_i,
            rst_n_i          =>  rst_n_i,
            start_i          =>  vr_start,
            row_idx_i        =>  max_idx_r,
            a_i              =>  a_i,
            a_addr_o         =>  vr_a_addr,
            b_i              =>  b_i,
            b_addr_o         =>  vr_b_addr,
            z_o              =>  vr_z,
```

```
            z_addr_o        =>  vr_z_addr,
            wr_o            =>  vr_wr,
            done_o          =>  vr_done
        );

    EH: e_by_Htr
        port map (
            clk_i           =>  clk_i,
            rst_n_i         =>  rst_n_i,
            start_i         =>  eh_start,
            vector_base_o   =>  eh_vec_base,
            block_base_o    =>  eh_blk_base,
            result_base_o   =>  eh_res_base,
            start_mul_o     =>  eh_start_mul,
            mul_done_i      =>  vc_done,
            done_o          =>  eh_done
        );

    NS: null_syndrome
        port map (
            clk_i           =>  clk_i,
            rst_n_i         =>  rst_n_i,
            start_i         =>  ns_start,
            a_i             =>  a_i,
            a_addr_o        =>  ns_a_addr,
            b_i             =>  b_i,
            b_addr_o        =>  ns_b_addr,
            z_o             =>  ns_z,
            z_addr_o        =>  ns_z_addr,
            wr_o            =>  ns_wr,
            null_syn_o      =>  ns_null_syn,
            done_o          =>  ns_done
        );

    CT: clear_temp
        port map (
            clk_i           =>  clk_i,
            rst_n_i         =>  rst_n_i,
            start_i         =>  ct_start,
            a_i             =>  a_i,
            a_addr_o        =>  ct_a_addr,
            b_i             =>  b_i,
            b_addr_o        =>  ct_b_addr,
            z_o             =>  ct_z,
            z_addr_o        =>  ct_z_addr,
            wr_o            =>  ct_wr,
            done_o          =>  ct_done
        );

    CM: comp_message
        port map (
            clk_i           =>  clk_i,
            rst_n_i         =>  rst_n_i,
            start_i         =>  cm_start,
            a_i             =>  a_i,
            a_addr_o        =>  cm_a_addr,
            b_i             =>  b_i,
            b_addr_o        =>  cm_b_addr,
            z_o             =>  cm_z,
            z_addr_o        =>  cm_z_addr,
            wr_o            =>  cm_wr,
            done_o          =>  cm_done
```

```
        );

    seq: process(clk_i, rst_n_i)
    begin
        if rst_n_i = '0' then
            state       <= IDLE;
            max_idx_r   <= (others => 0);
            l_r         <= 0;
        elsif rising_edge(clk_i) then
            state       <= next_state;
            max_idx_r   <= max_idx_c;
            l_r         <= l_c;
        end if;
    end process seq;
    -- TODO: put everything in sensitivity list
    comb: process(all)
--state, start_i,
--          kr_a_addr, kr_b_addr, kr_z, kr_z_addr, kr_wr, kr_done,
--          vc_a_addr, vc_b_addr, vc_z, vc_z_addr, vc_wr,
--          xl_start_mul, xl_vec_base, xl_blk_base, xl_res_base, xl_done,
--          hs_start_mul, hs_vec_base, hs_blk_base, hs_res_base, hs_done,
--          qs_start_mul, qs_vec_base, qs_blk_base, qs_res_base, qs_done,
--          fm_a_addr, fm_b_addr, fm_z, fm_z_addr, fm_wr, fm_max_idx, fm_done,
--          vr_a_addr, vr_b_addr, vr_z, vr_z_addr, vr_wr, vr_done,
--          eh_start_mul, eh_vec_base, eh_blk_base, eh_res_base, eh_done)
    begin
        next_state      <= state;
        a_addr_o        <= to_unsigned(0, address_bits);
        b_addr_o        <= to_unsigned(0, address_bits);
        z_o             <= to_unsigned(0, position_bits);
        z_addr_o        <= to_unsigned(0, address_bits);
        wr_o            <= '0';
        failure_o       <= '0';
        done_o          <= '0';
        kr_start        <= '0';
        vc_start        <= '0';
        vc_binary       <= '0';
        vc_vec_base     <= to_unsigned(0, address_bits);
        vc_blk_base     <= to_unsigned(0, address_bits);
        vc_blk_weight   <= 0;
        vc_res_base     <= to_unsigned(0, address_bits);
        vc_op           <= '0';
        xl_start        <= '0';
        hs_start        <= '0';
        qs_start        <= '0';
        fm_start        <= '0';
        max_idx_c       <= max_idx_r;
        vr_start        <= '0';
        eh_start        <= '0';
        ns_start        <= '0';
        ct_start        <= '0';
        cm_start        <= '0';
        l_c             <= l_r;
        case state is
            when IDLE =>
                l_c <= 0;
                if start_i = '1' then
                    next_state  <= COMPUTE_LTR;
                end if;
            when COMPUTE_LTR =>
                a_addr_o    <= kr_a_addr;
                b_addr_o    <= kr_b_addr;
```

```
                z_o         <=  kr_z;
                z_addr_o    <=  kr_z_addr;
                wr_o        <=  kr_wr;
                kr_start    <=  '1';
                if kr_done = '1' then
                    next_state  <=  COMPUTE_INITIAL_SYNDROME;
                end if;
            when COMPUTE_INITIAL_SYNDROME =>
                a_addr_o        <=  vc_a_addr;
                b_addr_o        <=  vc_b_addr;
                z_o             <=  vc_z;
                z_addr_o        <=  vc_z_addr;
                wr_o            <=  vc_wr;
                vc_start        <=  xl_start_mul;
                vc_binary       <=  '1';
                vc_vec_base     <=  xl_vec_base;
                vc_blk_base     <=  xl_blk_base;
                vc_blk_weight   <=  sum(M)*DV;
                vc_res_base     <=  xl_res_base;
                vc_op           <=  '0';
                xl_start        <=  '1';
                if xl_done = '1' then
                    next_state  <=  COMPUTE_SIGMA;
                end if;
            when COMPUTE_SIGMA =>
                a_addr_o        <=  vc_a_addr;
                b_addr_o        <=  vc_b_addr;
                z_o             <=  vc_z;
                z_addr_o        <=  vc_z_addr;
                wr_o            <=  vc_wr;
                vc_start        <=  hs_start_mul;
                vc_binary       <=  '0';
                vc_vec_base     <=  hs_vec_base;
                vc_blk_base     <=  hs_blk_base;
                vc_blk_weight   <=  DV;
                vc_res_base     <=  hs_res_base;
                vc_op           <=  '1';
                hs_start        <=  '1';
                if hs_done = '1' then
                    next_state  <=  COMPUTE_R;
                end if;
            when COMPUTE_R =>
                a_addr_o        <=  vc_a_addr;
                b_addr_o        <=  vc_b_addr;
                z_o             <=  vc_z;
                z_addr_o        <=  vc_z_addr;
                wr_o            <=  vc_wr;
                vc_start        <=  qs_start_mul;
                vc_binary       <=  '0';
                vc_vec_base     <=  qs_vec_base;
                vc_blk_base     <=  qs_blk_base;
                vc_blk_weight   <=  qs_blk_weight + 1;
                vc_res_base     <=  qs_res_base;
                vc_op           <=  '1';
                qs_start        <=  '1';
                if qs_done = '1' then
                    next_state  <=  FIND_B;
                end if;
            when FIND_B =>
                a_addr_o        <=  fm_a_addr;
                b_addr_o        <=  fm_b_addr;
                z_o             <=  fm_z;
```

```
                z_addr_o        <=  fm_z_addr;
                wr_o            <=  fm_wr;
                fm_start        <=  '1';
                if fm_done = '1' then
                    max_idx_c   <=  fm_max_idx;
                    next_state  <=  COMPUTE_ERROR;
                end if;
            when COMPUTE_ERROR =>
                a_addr_o        <=  vr_a_addr;
                b_addr_o        <=  vr_b_addr;
                z_o             <=  vr_z;
                z_addr_o        <=  vr_z_addr;
                wr_o            <=  vr_wr;
                vr_start        <=  '1';
                if vr_done = '1' then
                    next_state  <=  COMPUTE_SYNDROME;
                end if;
            when COMPUTE_SYNDROME =>
                a_addr_o        <=  vc_a_addr;
                b_addr_o        <=  vc_b_addr;
                z_o             <=  vc_z;
                z_addr_o        <=  vc_z_addr;
                wr_o            <=  vc_wr;
                vc_start        <=  eh_start_mul;
                vc_binary       <=  '1';
                vc_vec_base     <=  eh_vec_base;
                vc_blk_base     <=  eh_blk_base;
                vc_blk_weight   <=  sum(M)*DV;
                vc_res_base     <=  eh_res_base;
                vc_op           <=  '0';
                eh_start        <=  '1';
                if eh_done = '1' then
                    next_state  <=  CHECK_SYNDROME;
                end if;
            when CHECK_SYNDROME =>
                a_addr_o        <=  ns_a_addr;
                b_addr_o        <=  ns_b_addr;
                z_o             <=  ns_z;
                z_addr_o        <=  ns_z_addr;
                wr_o            <=  ns_wr;
                ns_start        <=  '1';
                if ns_done = '1' then
                    if ns_null_syn = '1' or l_r >= 20 then
                        next_state  <=  COMPUTE_MESSAGE;
                    else
                        next_state  <=  CLEAR_TEMP_AND_LOOP;
                    end if;
                end if;
            when CLEAR_TEMP_AND_LOOP =>
                a_addr_o        <=  ct_a_addr;
                b_addr_o        <=  ct_b_addr;
                z_o             <=  ct_z;
                z_addr_o        <=  ct_z_addr;
                wr_o            <=  ct_wr;
                ct_start        <=  '1';
                if ct_done = '1' then
                    l_c         <=  l_r + 1;
                    next_state  <=  COMPUTE_SIGMA;
                end if;
            when COMPUTE_MESSAGE =>
                a_addr_o        <=  cm_a_addr;
                b_addr_o        <=  cm_b_addr;
```

```
                z_o             <=  cm_z;
                z_addr_o        <=  cm_z_addr;
                wr_o            <=  cm_wr;
                cm_start        <=  '1';
                if cm_done = '1' then
                    next_state  <=  CLEAR_TEMP_AND_RETURN;
                end if;
            when CLEAR_TEMP_AND_RETURN =>
                a_addr_o        <=  ct_a_addr;
                b_addr_o        <=  ct_b_addr;
                z_o             <=  ct_z;
                z_addr_o        <=  ct_z_addr;
                wr_o            <=  ct_wr;
                ct_start        <=  '1';
                if ct_done = '1' then
                    next_state  <=  DONE;
                end if;
            when DONE =>
                done_o  <=  '1';
                if l_r >= 20 then
                    failure_o   <=  '1';
                end if;
                if start_i = '0' then
                    next_state  <=  IDLE;
                end if;
            when others =>
                null;
        end case;
    end process comb;

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, 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, rev. 25 November 2018