Towards practical code based signature: Implementing fast and compact QC-LDGM signature scheme on embedded hardware

Jingwei Hu, Ray C.C. Cheung, *Member, IEEE*

Abstract—In this paper, we present fast and yet compact implementations for code based signature. Existing designs either require enormous memory storage or commit very slow issuing speed of signatures. The solution we propose here solves these problems by exploiting QC-LDGM code at different levels. In particular, we propose a new signature issuing algorithm and give detailed and optimized solutions for critical steps of this algorithm. For the time being, the design presented in this work is the fastest implmentation of code based signature in public publications. We show for instance that our implementation of signature signing engine can sign approximately 100,000 signatures per second on a Xilinx Virtex-6 FPGA, requiring only 7126 slices and 60 memory blocks. On the contrary, we also provide a very compact implementation which can sign 5681 signatures per second with only 18 memory blocks.

Index Terms—Code-based Cryptography, Niederriter encryption Scheme, QC-MDPC Code, FPGA Implementation.

1 Introduction

Modern public-key cryptographic systems rely on either the integer factorization or discrete logarithm problem, both of which would be easily solvable on large enough quantumn computers using Shor's algorithm [1]. The upcoming breakthroughs of powerful quantum computers have shown their potential in computing solutions to the problems mentioned above [2], [3]. The cryptographic research community has recognized the urgency of this challenge and begun to settle their security on alternative hard problems in the last years, such as multivariate-quadratic, lattice-based and code-based cryptosystems [4]. We address in this paper the problem of implementing code-based signature on small embedded hardware, or in other terms, the problem of producing signatures fast with very limited memory quota. This is of interest, in particular, if we wish to replace the currently used RSA or ECC with code based cryptosystem when commercial quantum computers are powerful enough to attack them.

The Courois-Finiasz-Sendrier (CFS) scheme [5] and the Kabatianskii-Krouk-Smeets (KKS) scheme [6], [7], [8] are two main code based signature proposals. The KKS scheme choose two codes with different size to create the trapdoor but an important weakness of this scheme was recently pointed out in [9]. The CFS scheme instead use almost complete decoding to decode the hash value of the message. The idea is that, for example, assume the hashed document h is the syndrome of an erroneous codeword that corresponds to the error vector with $t + \delta$ errors but the error control code CFS exploits, *i.e.* classical Goppa code in Niederreiter's dual form [10] can only correct t errors, the system randomly reverse the values in δ positions of the error vector followed by decoding it. h would become decodable within finite iterations. The main drawback of the CFS scheme is the tedious long signing time. In the original proposal, approximately 9!=362,880 decoding attempts are required before producing a valid signature. Recently, the so-called DOOM-GBA attack [11] further reduced the time complexity of the signature forgery. M. Finiasz proposed the countermeasure parallel-CFS [12] to resist this attack. The idea is to produce $\lambda = 3, 4$ different hash values from the same document and sign each of them separately. Regarding the timing performance of this new CFS, it is even worse with a factor of λ compared with the classical CFS.

Baldi *et. al* propose a new solution [13] in which they consider only a subset of the possible syndromes and replace traditional Goppa codes with low-density generator matrix (LDGM) codes [14], [15], [16], [17]. This way, they obtain a considerable reduction in the public key size. In addition, the complicated syndrome decoding through the private code is also simplified to a straightforward procedure.

The first and only hardware implementation of CFS scheme that we are aware of is available in [18], reporting an average of 0.86 second of signature issuing on a Xilinx SCV300E FPGA. A software implementation of parallel-CFS [11], [19] is conducted on Intel Xeon W3680 running at 3.20GHz. For 80-bit security, their experiment results report 1.32 seconds of running time with public key size equal to 20 Mb. The parallel-CFS signature speed record holder, Mcbits [20] published in 2013 issues a signature in estimated 0.156 second, implemented on a single core of Intel Core i5 processor.

The purpose of this work is to implement a practical code based signature for embedded hardware, that is fast and compact for real-world applications. Current implementations are all CFS or parallel-CFS signature, which is slow speed and has large key size. Our work instead is based on LDGM code. The contributions of this paper include:

J. Hu and R. Cheung are with the Department of Electronic Engineering, City University of Hong Kong, Hong Kong e-mail: j.hu@my.cityu.edu.hk; r.cheung@cityu.edu.hk.
Manuscript received XX XX, 2016; revised XX XX, 2016.

1) We propose a new QC-LDGM code signature issuing algorithm with smaller secret key size and fast speed. This method allows us to implement a fast yet compact signature signing engine.
2) We address the problem of efficiently implementing the orthogonal function $\mathcal{F}_\theta$. Our new proposal exploits and improves Sendrier's constant weight coding [21].
3) We address the problem of storing large secret keys by exploiting the sparse nature of LDGM code. For 80-bit security, our new method reduces this overhead by a factor of 7.4.
4) Furthermore, we implement a new code based signature signing and verifying engine on FPGA platform, based on the above proposals. In particular, we give a fast signature signing implementation which takes $10\mu s$ per signature, and another compact implementation which takes $176\mu s$ but consumes only 18 block memories. At this moment, our implementation holds the speed record of code based signature.

The remaining part of this paper is organized as follows. We briefly revisit the QC-LDGM code signature proposed by Baldi *et. al* [13] in Section 2. This motivates us to propose a more efficient signature generation algorithm and to make proper decisions for our implementations, presented in Section 3. Section 4 describes our actual FPGA implementation of the proposed signature generator and verifier. The experimental results we have collected are presented in Section 5. Section 6 concludes our work.

2 QC-LDGM Code Signature Scheme

Baldi *et. al* proposed in 2013 a novel code based digital signature scheme using LDGM codes and sparse syndromes [13]. To further reduce memory storage for the public/private keys, they also suggested to apply its quasi-cyclic variant — QC-LDGM code $C(n, k, p)$ for this signature scheme. The authors also proposed sets of code parameters for three different security levels, shown in Table 1. Hereafter we denote quasi-cyclic (QC) form of a generator matrix G as G_{QC}:

$$G_{QC} = \begin{bmatrix} G_{0,0} & G_{0,1} & \cdots & G_{0,n_0-1} \\ G_{1,0} & G_{1,1} & \cdots & G_{1,n_0-1} \\ \vdots & \vdots & \ddots & \vdots \\ G_{k_0-1,0} & G_{k_0-1,1} & \cdots & C_{k_0-1,n_0-1} \end{bmatrix} \quad (1)$$

where $G_{i,j}$ represents a sparse circulant matrix or a null matrix with size $p \times p$. Hence we have $k_0 \times n_0$ circulant blocks in G_{QC} where $k_0 = k/p$ and $n_0 = n/p$. Likewise, when we compute the null space of the generator matrix G_{QC}, we also obtain the $n \times r$ parity check matrix in QC form H_{QC} with $n_0 \times r_0$ circulant blocks in total, $r_0 = r/p = (n-k)/p$. The benefit of such QC construction comes from a dramatic reduction of memory, as we simply store the first row/column of each circulant block $G_{i,j}$, the remaining part can be recovered by cyclic shift of this row/column, which precisely reduces to $1/p$ times of the matrix size.

The key generation part is described in Algorithm 1. The starting point of the key generation is to produce the $n \times r$ systematic parity check matrix H_{QC} of QC-LDGM code, with an identity block in the rightmost part. It can be obtained by first constructing the $k \times n$ generator matrix with two block matrices such that $G_{QC} = [G_0|G_1]$ and then calculating $H = [(G_0^{-1}G_1)^T|I]$ where $(\cdot)^T$ denotes matrix transpose and I denotes the identity matrix. It is worth noting that not every G_0 out of G is invertible but we can perform Gauss-Jordan elimination to put G_{QC} in such a form that G_0 is invertible. The matrix S_{QC} is a sparse non-singular matrix in QC form, with average row and column weight $m_S \ll n$. The matrix Q_{QC} is called a weight controlling transformation matrix defined in [13], [17]. We need this transformation matrix because we must preserve the sparsity of a sparse vector during signature producing when imposing a linear transformation on it. In other terms, given a sparse vector s, we can ensure the $s' = Q_{QC} \cdot s$ is also a sparse vector. This kind of matrices can be obtained as $Q_{QC} = R_{QC} + T_{QC}$ where R_{QC} is a $r \times r$ low rank dense matrix and T_{QC} is a sparse matrix with average row and column weight $m_T \ll n$. Note that $R_{QC} = (a_{r_0}^T \cdot b_{r_0}) \bigotimes \mathbf{1}_{p\times p}$ and $(b_{r_0} \bigotimes \mathbf{1}_{p\times p}) \cdot s = \mathbf{0}_{z\times 1}$ where $z < r$ and $\bigotimes$ denotes Kronecker product. This way, we can verify that $R_{QC} \cdot s = \mathbf{0}_{r\times 1}$ and $s' = Q_{QC} \cdot s = T_{QC} \cdot s$. Hence, the Hamming weight of s' is at most equal to m_T times that of s.

Algorithm 1: Key Generation of QC-LDGM code signature

Input: n, p, k, w, w_g, z, m_T, m_s
Output: public/private key pair
1 Randomly generate LDGM code $C(n, k, p)$ generator matrix G_{QC} in QC form with row hamming weight w_g
2 Obtain the parity check matrix H_{QC} in systematic form from G_{QC}
3 Randomly generate two non-singular matrices: Q_{QC} and S_{QC} according to parameters w, z, m_T, m_s
4 The public key is obtained as $H'_{QC} = Q_{QC}^{-1} \cdot H_{QC} \cdot S_{QC}^{-1}$
5 **return** H'_{QC} *as public key and* H_{QC}, Q_{QC}, S_{QC} *as private key*

Algorithm 2 describes how to produce a unique digital signature from some document M. For simplicity, we have assumed the hash value of M is acquired. The most critical step is to find a sparse vector $s_{r\times 1}$ with Hamming weight w such that $b = b_{r_0} \bigotimes \mathbf{1}_{p\times p}$ is orthogonal to $s_{r\times 1}$. The authors proposed a framework to implement such function $\mathcal{F}_\Theta$ as follows [13]. Given a message digest h of length x bits, append it with y-bit value of a counter, thus obtaining $[h|l]$. The value of $[h|l]$ is then mapped uniquely into a r-bit vector of weight w. The counter is initially set to zero and then progressively increased until a r-bit vector $s_{r\times 1}$ is found orthogonal to b. However, the authors did not mention how to configure the bit length of such digest and counter for an optimal performance of signature producing. In addition, we observe some redundancy in Algorithm 2 and propose an improved version of it, allowing for a faster yet more memory efficient implementation. We would give a thorough discussion on this topic in Section 3.

TABLE 1: System parameters for some security levels (SL) of the LDGM code signature scheme, referenced from [13]

SL (bits)	n	k	r	p	w	w_g	w_c	z	m_T	m_S	S_k* (Kbits)
80	9800	4900	4900	50	18	20	160	2	1	9	938
120	24960	10000	14960	50	23	25	325	2	1	14	7293
160	46000	16000	30000	50	29	31	465	2	1	20	26953

*public key H'_{QC}, an n by r quasi-cyclic matrix over $GF(2)$.

Algorithm 2: Signature generation of QC-LDGM code signature

Input: message digest h, orthogonal function $\mathcal{F}_\Theta(\cdot)$, secret key ($H_{QC}$, Q_{QC}, S_{QC})
Output: public signature e'

Compute $s = \mathcal{F}_\Theta(h)$ such that $(b_{r_0} \bigotimes \mathbf{1}_{p\times p}) \cdot s = \mathbf{0}_{z\times 1}$ verifies, as key generation part mentions.
Compute the private syndrome $s' = Q_{QC} \cdot s$
Compute the error vector $e = [\mathbf{0}_{1\times k} | s'^T]$
Randomly pick up a codeword c with weight w_c of QC-LDGM code $C(n, k)$
return $e' = (e + c) \cdot S_{QC}^T$

After receiving the message hash, the digital signature and the public orthogonal function $\mathcal{F}_\Theta(\cdot)$, the verifier computes the syndrome $\hat{s} = \mathcal{F}_\Theta(h)$ and $H'_{QC} \cdot e'^T = Q_{QC}^{-1} \cdot H_{QC} \cdot S_{QC}^{-1} \cdot S_{QC} \cdot (e^T + s^T) = Q_{QC}^{-1} \cdot H_{QC} \cdot e^T = Q_{QC}^{-1} \cdot s' = s$. If $s = \hat{s}$, the signature is accepted, otherwise it is discarded. Prior to this operation, the verifier can check whether the Hamming weight of e' is at most $(m_T w + w_c)m_s$ and that of $\hat{s}$ is w to make a correct decision as early as possible. The verification process is included in Algorithm 3.

Algorithm 3: Signature verification of QC-LDGM code signature

Input: digital signature e', message digest h, orthogonal function $\mathcal{F}_\Theta(\cdot)$, public key H'_{QC}
Output: signature verified or not

if *weight of e' is larger than* $(m_T w + w_c)m_s$ **then**
  return *False*
else
  $\hat{s} = \mathcal{F}_\Theta(h)$
  if *the hamming weight of $\hat{s}$ is not w* **then**
    return *False*
  else
    compute $s = H'_{QC} \cdot e'^T$
    if $s == \hat{s}$ **then**
      return *True*
    else
      return *False*

3 Design Decisions for Embedded Hardware

In this section, we discuss our decisions on implementing the QC-LDGM code signature on small, embedded systems. In particular, we give a detailed description of the orthogonal function $\mathcal{F}_\Theta$ implementation, how to pick up a random codeword c, how we exploit the sparse syndromes to speed up the signature producing and an efficient solution to store large private keys. Our prototype platform is Xilinx Virtex-6 device.

3.1 Implementing $\mathcal{F}_\Theta$ using constant weight coding

The purposes of the orthogonal function $\mathcal{F}_\Theta$ of signature generation include:

1) Convert the input hash value h uniquely into a r-bit vector s with weight of w.
2) s must be orthogonal to the vector $b = b_{r_0} \bigotimes \mathbf{1}_{p\times p}$ where b_{r_0} is a randomly generated $z \times r_0$ matrix and $\mathbf{1}_{p\times p}$ is a $p \times p$ all-one matrix.

The first one refers to the constant weight coding, that is, the problem of encoding information into binary words of prescribed length and Hamming weight. The exact solution [22] of constant weight coding requires the computation of large binomial coefficients and has a quadratic complexity though it is optimal as a source coding algorithm. Sendrier later proposed an elegant solution [21] with a linear complexity and has a high coding efficiency very close to 1 by exploiting Goloumb's run-length encoding method [23]. His proposal is fairly easy to implement but the most critical step of computing the value of d requires dedicated float point unit which is not an easy job for resource constrained hardwares. Heyse *et al.* later solved this problem by using a small look up table to compute the value of d for constant weight encoding [24], [25] when they implemented Niederreiter encryption scheme [10] on embedded micro-controllers and FPGAs. Nevertheless, we observe that 1) their lookup table of pre-stored data has space complexity of $\mathcal{O}(n)$, which is still quite big for small embedded systems making their design less unscalable if n is large. 2) On top of this, the coding efficiency (average number of bits read for a successful encoding) drops rapidly as n increases.

We thus figure out a new way of computing the value of d to solve these two drawbacks based on Heyse's work, shown in Algorithm 4. Assume we encode the input into a r-bit word with w 1's, exactly as $\mathcal{F}_\Theta(\cdot)$ requires, we propose to compute $d = r \times \theta[w]$ where $\theta[w] = 1 - 1/2^{\frac{1}{w}}$. $\theta[w]$ is a pre-computed fixed point number and stored in a lookup table as w is a very small integer. Then d can be directly computed using an integer multiplier. Note that $\theta[w]$ is actually fractional and how many bits we use to store this fractional number, *i.e.* the precision of $\theta[w]$ significantly affects the coding performance. We therefore run experiments with 40,000 message samples to determine the optimal precision of this fractional number. Experiments reveal that 5 bit of precision is the best for 80-bit and 120-bit security whereas

TABLE 2: The performance of our proposed CW coding method, using parameters recommended in LDGM code signature scheme

(r, w)	SL	$\theta[w]$ precision$_*$ (bits)	Average number of bits read	Upper bounded entropy$_\dagger$ $log_2\binom{r}{w}$ (bits)	Prestored data for $\theta[w]$ (bits)
$(4900, 18)$	80 bit	5	165.68	168.10	40
$(14960, 23)$	120 bit	5	241.45	244.51	54
$(30000, 29)$	160 bit	6	324.63	328.49	54

*The number of bits that we use to represent the fractional number $\theta[w] = 1 - 1/2^{\frac{1}{w}}$.

†The average number of bits we read for encoding a constant weight word is upper bounded by the entropy of the constant weight word $W_{r,w}$ equipped with a uniform distribution $H(W_{r,w}) = log_2\binom{r}{w}$. One can observe that the coding efficiency is close to 1 when using our method.

6 bit is the optimal one for 160-bit SL. The results are all collected in Table 2.

Algorithm 4: Proposed constant weight encoding method (Bin2CW)

Input: message length r, message weight w and a binary stream B

Output: a w-tuple $I = (i_1, i_2, \ldots, i_w)$, indicating the indexes of the 1's in the constant weight word

$\delta = 0, i_0 = -1, index = 1$
while $w > 0$ **do**
  if $r \leq w$ **then**
    $w{-}{-}, r{-}{-}$
    $i_{index} = i_{index-1} + \delta + 1$
    $\delta = 0, index{+}{+}$
  else
    $d = r \times \theta[w]$
    if *read*$(B, 1) = 1$ **then**
      $r{-}{=} d$, $\delta{+}{=} d$
    else
      $i = \text{read}(B, \lceil log_2(d) \rceil)$
      $\delta{+}{=} i$
      $i_{index} = i_{index-1} + \delta + 1$
      $r{-}{=} (i + 1), w{-}{-}, \delta = 0, index{+}{+}$
return $(i_1, \ldots, i_w)$

The second problem is to ensure that the produced constant weight vector s is orthogonal to b. To check the orthogonality, or we call it the orthogonality test, is easy on hardware. We first obtain the indexes i's of the non-zero entries of s by CW coding, then we sum the columns $b[:, i]$ according to these indexes. If the result is zero, then s is orthogonal to b. Unfortunately, it is very unlikely that s is orthogonal to b because the CW coding maps the input message digest h uniquely to s but s is not necessarily orthogonal to the public known b. Authors in [13] suggested to append a counter l of y-bits to h of x-bits to make the inputs $[h|l]$ invariable such that CW coding produces different constant weight vector s for the same message digest h. By trials and errors one can always find such vector s that satisfies $b \cdot s = 0$. Here in this paper, we accept this framework and elaborate two configurations for our hardware implementations:

- How to configure the length of counter l, or in other terms, how to determine the value of y ?
- How many bits (x-bits) of message digest h should commit for the CW encoding ?

TABLE 3: $[h|l]$ configurations for embedded sytems

(r, w)	SL	bits of $[h|l]$	x bits of h	y bits of l	failure rate
$(4900, 18)$	80 bit	186	180		
$(14960, 23)$	120 bit	262	256	6	1.009×10^{-8}
$(30000, 29)$	160 bit	345	339		

The first configuration is closely related to the failure rate of the orthogonality test. The length of counter l restricts the maximum number of orthogonality test trials, that is 2^y. Hence the problem is simplified to determine the value of y, such that the failure rate of 2^y independent orthogonality tests is negligible. Note that b is a $z \times r$ matrix and it is commonly constructed by a good uniform random number generator. Then the probability that the constant weight word s passes a single orthogonality test is:

$$Prob(s \text{ is orthogonal to } b) = \left(\frac{\binom{w}{0} + \binom{w}{2} + \cdots + \binom{w}{2k}}{2^w}\right)^z = (\frac{2^{w-1}}{2^w})^z = \frac{1}{2^z} \quad (2)$$

where $k = \lfloor \frac{w}{2} \rfloor$. Using Equation (2), we obtain the failure rate of 2^y independent orthogonality tests is:

$$Prob(\text{all orthogonality tests fails}) = (1 - \frac{1}{2^z})^{2^y} \quad (3)$$

The recommended value of z is 2 and we therefore decide to set $y = 6$ making the failure rate equal to 1.009×10^{-8}.

The second configuration concerns the invariable coding length of CW encoding. For different input message digest h, the number of bits CW encoder reads is also different. Hence we must give a upper bound of the bits read for CW coding in order to set up a correct x value. Authors can verify that this upper bound is met when we input all-zero binary inputs. For the 80-bit, 120-bit and 180-bit recommended parameters, this upper bound is 180 bits, 256 bits and 339 bits respectively. The detailed configurations of x and y are summarized in Table 3.

One last thing worth mentioning is that the order of h and l must be carefully chosen. If we append l at the back of h, it could be problematic as the binary string $[h|l]$ only changes its last 6 bits during each orthogonality test iteration but as a consequence, the produced constant weight vector s also changes its last few bits or even worse, does not change at all. This order definitely increases the failure rate of the tests, and we therefore suggest to put l at

the beginning or randomly insert the 6 bits of l into h. In our practical implementations, we set it to be $[l|h]$.

3.2 Generating a random codeword c by LCG-based pseudorandom number generator

In the process of signature generation, a random codeword c is picked up to resist signature forgery attacks [13]. The authors in [13] also mentioned that such codeword c should be chosen by a deterministic function of the document M and hence, of the public syndrome s. To achieve this, we use a pseudorandom number generator (PRNG) based on linear congruent generation to extract indexes of rows of the generator matrix G and then sum them to produce c. In particular, our implementation accepts parameters from Microsoft Visual Basic PRNG, defined by a linear recurrence as follows:

$$X_{n+1} = aX_n + c \bmod m \tag{4}$$

where a is set to 1140671485, c set to 12820163 and m set to 2^{24}. The initial state X_0 is set to $[l|h]$ to guarantee the sequence of X_i's is deterministic. LCG is fairly easy to implement on small embedded systems, with a single constant-coefficient multiplier in which we set the constant to be 1140671485.

Note that signature signing algorithm requires that the randomly generated c has a constant weight of w_c. We can instead use w_c/w_g distinct X_i's to extract w_c/w_g rows of G_{QC} for generating such c. w_g is the Hamming weight of a row of G_{QC}, much less than n and hence it is with high probability that the sum of distinct w_c/w_g rows of G_{QC} has weight close to w_c. Even if c that we produce happens to have weight smaller than this range, though the probability is low, it can pass the signature verification steps in Algorithm 3 since e' is still smaller than $(m_T w + w_c)m_s$ with such c.

3.3 Exploiting the sparse syndromes to accelerate computations of signature issuing

The signature verification is relatively simple because the main step is to do a matrix multiplication $H'_{QC} \cdot e'^T$ whereas things becomes complicated when signing a signature: we have to do more matrix multiplications, for instance, to test the orthogonality by computing $b \cdot s$, to obtain the secret syndrome $s' = Q_{QC} \cdot s$, to generate a random codeword $c = u \cdot G_{QC}$ (Let u denote the output sequence of PRNG) and finally to produce the signature by multiplying the scramble matrix S_{QC}. The problem though is, these matrix multiplications deal with very large operands and thus are really time-consuming. In addition, the different operand sizes of these multiplications also make it difficult to handle on hardware.

In order to solve this problem making it suitable for hardware implementations, we propose a new framework of the signature generation based on Algorithm 2. The basic idea of this improvement comes from the observation that step 2—step 5 of the original proposal can be merged into a single operation. Note that $s' = Q_{QC} \cdot s = T_{QC} \cdot s$ and let $S_{QC} = [S_0|S_1]$, then we have signature rewritten as follows:

$$\begin{aligned} e' &= (e + c) \cdot S_{QC}^T \\ &= ([\mathbf{0}|s'] + u \cdot G_{QC}) \cdot S_{QC}^T \\ &= [\mathbf{0}|s^T T_{QC}^T] \cdot [\frac{S_0^T}{S_1^T}] + u \cdot G_{QC} S_{QC}^T \\ &= s^T \cdot (S_1 T_{QC})^T + u \cdot G_{QC} S_{QC}^T \end{aligned} \tag{5}$$

This way, we keep the new secret key $(S_1 T_{QC})^T$ and $G_{QC} S_{QC}^T$ to generate the signature e' by only two vector-matrix multiplications. Algorithm 5 describes our new algorithm.

Algorithm 5: Proposed signature generation of QC-LDGM code signature

Input: message digest h, orthogonal function $\mathcal{F}_\Theta(\cdot)$, secret key $A_{QC} = (S_1 T_{QC})^T$,$B_{QC} = G_{QC} S_{QC}^T$
Output: public signature e'

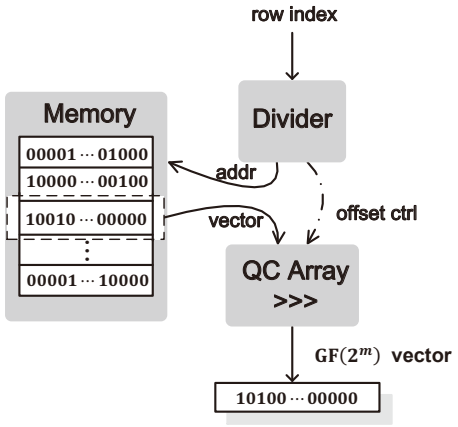
1 Compute $s = \mathcal{F}_\Theta(h)$ such that $(b_{r_0} \bigotimes \mathbf{1}_{p \times p}) \cdot s = \mathbf{0}_{z \times 1}$ verifies
2 Compute the first part $e_1 = s^T \cdot A_{QC}$
3 Produce a random sparse vector $u_{1 \times k}$ with weight $\frac{w_c}{w_g}$
4 Compute the second part $e_2 = u \cdot B_{QC}$
5 **return** $e' = e_1 + e_2$

It is worth mentioning that the above two vector-matrix multiplications can be done fast as the vector s and u is sparse. In our implementations, we record the indexes of the non-zero elements of s and u and extract the associated rows of the new secret key A_{QC}, B_{QC}. This way, multiplications are done by $w + w_c/w_g$ vector additions.

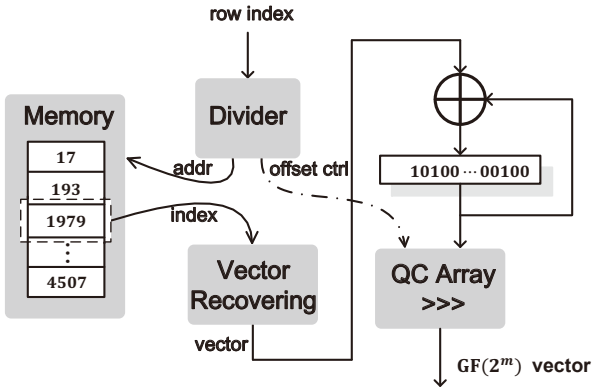
3.4 Managing the memory of private key efficiently

Memory management of the code based signature scheme is the most critical for embedded system implementations. This is because current available signature schemes [5], [11], [12], [13] usually have a considerably large key size whereas the memory storage on small embedded systems is very limited. [13] adopts the quasi-cyclic structure of the QC-LDGM code and reports a significant reduction of public key size H'_{QC}, as shown in Table 1. On the other hand, its secret key size that the authors did not mention about is larger as we have to store Q_{QC}, S_{QC}, G_{QC} and b, which actually are $\frac{r}{n} + \frac{k}{r} + \frac{n}{r} + \frac{z}{n} \approx 3.5$ times bigger than that of the secret key. For resource constrained embedded systems, this increase is indeed troublesome because the secret key size is already pushing the limit, for example, even for the lowest 80-bit security level, as large as 938 Kb and hence we need to store the 3.2 Mb of the public key, which is obviously a challenge for our systems. Comparatively, when using the proposed new secret keys in Algorithm 5, Section 3.3, this dense memory overhead is moderately reduced, with $\frac{n}{r} + \frac{z}{n} \approx 2$ times bigger than that of the public key.

Our first solution of memory management is quite straightforward and targets at fast speed performance of signature generation/verification. We store the first row of circulant blocks in A_{QC} and B_{QC} in block memory, and store the entire b. To retrieve a particular row of the QC



(a) Solution I, a straightforward implementation of extracting a row of QC matrices



(b) Solution II, a memory efficient implementation exploiting the sparse nature of secret keys

Fig. 1: The schematic of the proposed memory management solutions.

matrix, we first use an integer divider of small operands to compute the address of the first row of each circulant block in A_{QC}/B_{QC} and also the quasi-cyclic shift offset that should be performed. Then we extract this first row from the block memory according to the address we have just computed. Finally this first row is operated by a quasi-cyclic shifting array to recover the row we need. This work mechanism is illustrated in Figure 1 (a). Note that even a singe row of QC matrix is of thousands of bits but FPGA block memory resources do not have data width big enough to store one row at one address. This is the bottleneck for a high speed implementation. To improve the throughput of the memory reading, we decide to split the large data width into several smaller block memories that are supported by our FPGA device. Take 80-bit security parameters $n = 9800$ as an example, we use three memory blocks, each has 700-bit width. Therefore, it outputs 2100 bits at one time and it takes a total of 5 times to load the 9800 bits of one row completely. We do not use more memory blocks to further improve the throughput because the number of rows of Q_{QC}, S_{QC} is very small but Virtex-6 block memories cannot be configured to have such small data depth. More numbers of block memories result in a great waste of memory resources. For 80-bit security, the number of rows is 98, the data depth of 700-bit width memory we use is configured to 512. This way, we takes up the first 98*5 of 512 addresses, about 80% of utilization.

The second solution is much more memory efficient but the price we have to pay is the significant drop of timing performance. As aforementioned, many applications desire better memory utilization more than speed, it is reasonable that we put memory efficiency in the first place. The memory consumption shrinks if we observe that A_{QC} and B_{QC} are both sparse matrices, with average row weight equal to $m_A = m_S \cdot m_T$ and $m_B = m_S \cdot w_g$, respectively. We can instead trace the non-zero elements of the matrix and save their indexes to the memory. This way, we reduce the memory size to $\frac{m_A log_2 n}{n}$ and $\frac{m_B log_2 n}{n}$ of that of the first solution averagely. Note that the public key H'_{QC} is not a sparse vector and thus we cannot compress it using this method. In addition, we do not have to consider the memory width limit since we only process one index at a time and hence the data width is restricted to $log_2 n$ bits. This is also the reason why the timing performance is worse: a row has multiple non-zero elements and hence multiple indexes, and it takes much longer time to translate these indexes back into sparse vectors for further processing. We propose an efficient architecture to do this complicated index translation fast and we put the details in Section 4. Figure 1 (b) illusrates the mechanism of this solution. One index is read out from memory each time and translated into a $GF(2^m)$ vector with a single '1' in the designated position, by the index recovering unit. Several vectors like this are then summed to retrieve one particular row of the matrix. Such row vector is finally quasi-cyclic shifted to retrieve the row we are targeting at.

TABLE 4: Utilization report of the two proposed memory management

SL (bit)	Method	Secret key (Kb) A_{QC}	Secret key (Kb) B_{QC}	Public key (Kb) H'_{QC}
80	Method I	938	938	938
	Method II	12	241	
120	Method I	7293	4875	7293
	Method II	61	1025	
160	Method I	26953	14375	26953
	Method II	187.5	3100	

To summarize, Table 4 includes a detailed memory utilization comparison of both methods that we propose. Our first solution enables a fast implementation but also considers the time-area tradeoff. Our second solution targets at signing signatures with the lowest memory blocks, with acceptable timing performance. We would implement both solutions to demonstrate these points in the following sections.

4 Case Study: Proposed QC-LDGM signature architecture for 80 bit security

This section describes the proposed signature generator and the signature verifier of the 80-bit security level using the first parameter set listed in Table 1 [13]. Two different

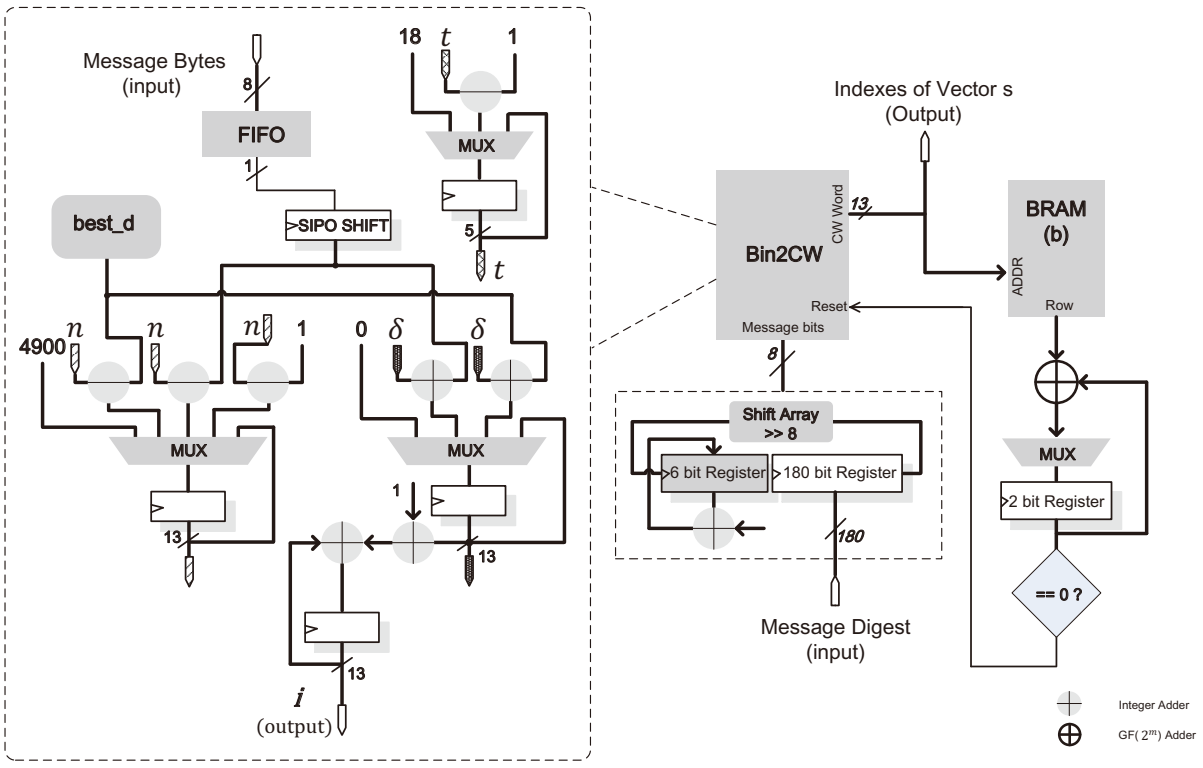


Fig. 2: Implementation of $\mathcal{F}_\theta$ function, using the constant weight encoder

signature generator architectures are proposed according to our proposed memory management methods, then the signature verifier handling the dense public key is also designed.

4.1 Signature Generator

The first architecture is shown in Figure 3 (a), the secret key A_{QC} and B_{QC} are stored in BRAMs, each has size of 98×9800 bits. However, Xilinx FPGA BRAM does not support a row of 9800 bits for reading, and we have to split such large data width into smaller pieces, in our cases, of 700 bits. As a result, a maximum of 2100 bits of data is read out and it takes five clock cycles to recover a specific row of A_{QC} or B_{QC}. Note that outcomes of $\mathcal{F}_\theta$ and PRNG are actually row indexes, and hence a radix-2 integer divider is used to translate every index into memory address and QC shift offset by dividing by 50. This divider is implemented by Xilinx FPGA IP core, with 15 clock cycles of delay. The correct row $A_{QC}[x,:]$ or $B_{QC}[x,:]$ associated with the index x is obtained by memory addressing and QC shift array. All these rows are summed to generate the signature.

The orthogonal function $\mathcal{F}_\theta$ (See Figure 2) is configured as described in Section 3.1. A 6-bit register is implemented as the counter and a 180-bit register holds the value of message digest. These two registers are conjugated and fed up into the Bin2CW byte by byte to produce the constant weight word. Whenever a CW word i becomes valid, the column of $b[:,i]$ is extracted and summed to the 2-bit register. After the last CW word is produced, the orthogonality is also verified by checking if the value of this register is zero. If the value is zero, we can output these CW words into a FIFO, otherwise we update the value of counter and reset the Bin2CW to produce a new series of CW words.

Figure 4 depicts the operation scheduling of the signature generator. $\mathcal{F}_\theta$ outputs the vector s to the FIFO and meanwhile, PRNG starts extracting rows of B_{QC} to generate e_2. Once the computation of e_2 is done, the signature generator triggers the processing of e_1 by extracting s from the FIFO because $e1$ and $e2$ share the same logic resources of vector addition and hence cannot be performed in parallel. The total delay is roughly the delay of e_1 plus that of e_2.

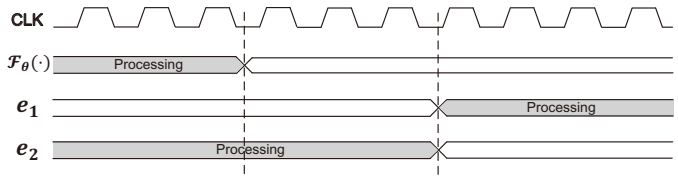
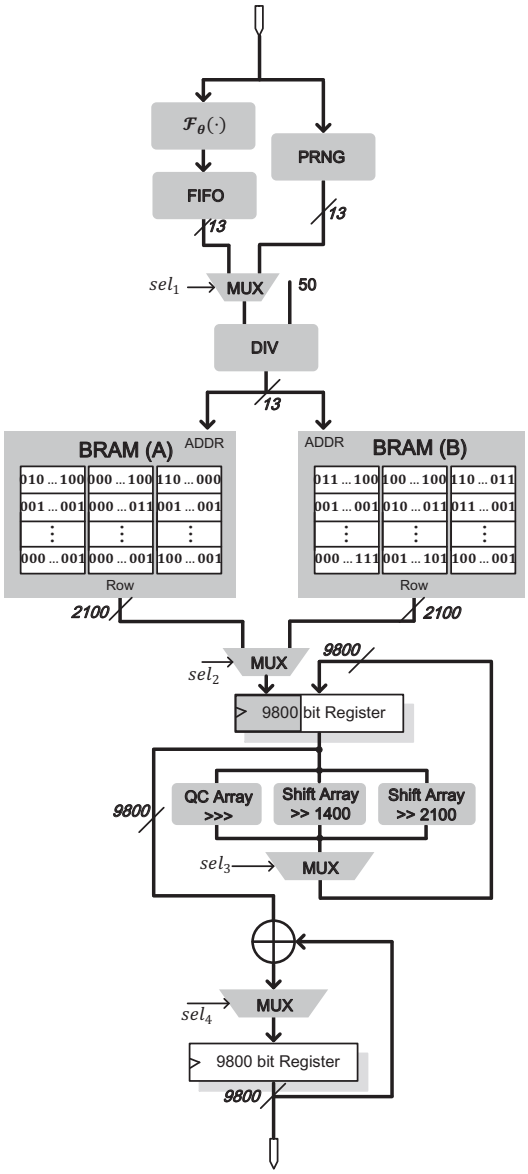
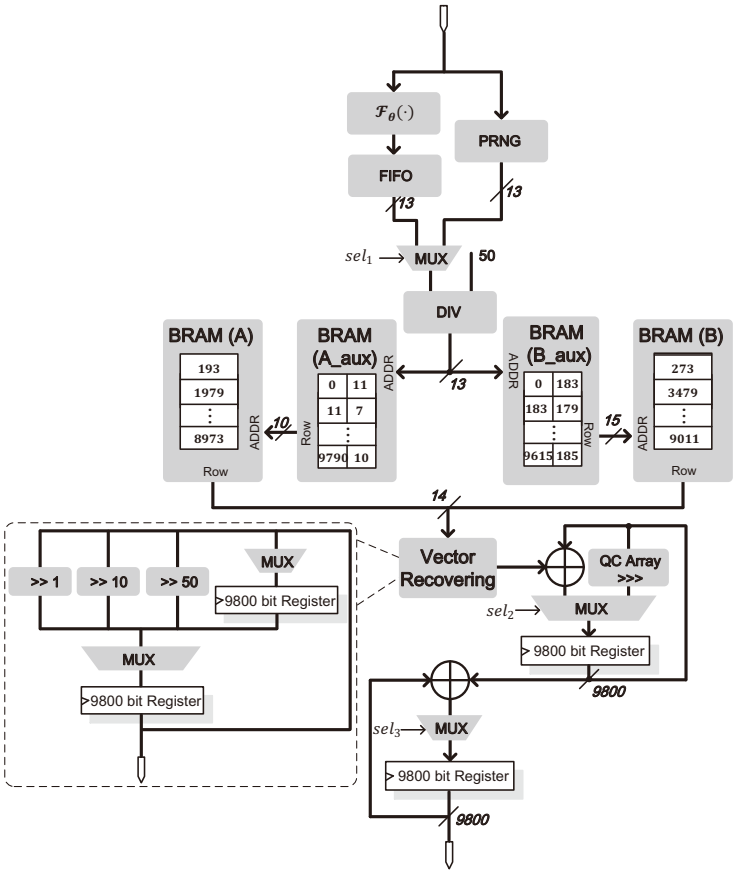


Fig. 4: The timing diagram of the proposed signature generator.

The second architecture is much more memory efficient, as we can see in Figure 3 (b). Indexes of all nonzero entries of A_{QC} and B_{QC} are stored in BRAM A and BRAM B, respectively. However, the Hamming weight of every row of A_{QC} or B_{QC} varies and thus we implement the additional A_aux and B_aux to keep these weight information and base addresses of A_{QC}, B_{QC}. This way, we exploit a two-level look up table to address a specific row of A_{QC} and B_{QC} by first looking up A_aux/B_aux and then extracting A/B. The data from A/B are processed by the vector recovering unit and summed up to constitute the correct row of A_{QC}



(a) a fast speed implementation of signature generator, using Solution I



(b) a memory efficient implementation of signature generator, using Solution II

Fig. 3: General architecture of signature generator for 80-bit security level

or B_{QC} we desire. The rest part of computations are similar to our first architecture. In fact, the timing diagram of our second architecture is identical to the first one except that the duration of e_2 is much longer than that of e_1.

It is worth mentioning that the operations of the vector recovering unit are extremely time-consuming: It takes around 9800 clock cycles to translate indexes of the same row of A_{QC}, B_{QC} into the associated 9800-bit sparse vector if we shift bit by bit. We must process 8 rows of A_{QC} and 18 rows of B_{QC}, and hence at least $254,800$ cycles of delay before producing the valid signature. To reduce such enormous delay, we propose to use multiple levels of shifting, which we call the 'big leap, small step' strategy, to accelerate this step. The idea is, we first use a shift w_1-bit ($w_1 > 1$) array to fast approximate the index i, and then shift the last $i \bmod w_1$ bits bit by bit. In practice, we use a more complicated three-level shifting $\langle w_1, w_2, 1\rangle$ ($w_1 > w_2 > 1$) to further improve the performance. Note that we need two extra constrains on w_1, w_2 for ease of our hardware implementations: 1) w_1 and w_2 must divide 9800. 2) w_2 must divide w_1. The row indexes of A_{QC}, B_{QC} are randomly generated and we therefore conduct a Monto-Carlo experiments with sample size equal to 100,000 to determine the optimal value of w_1, w_2, by evaluating all possible combinations of $\langle w_1, w_2, 1\rangle$. In our actual implementations we assume that both shifting w_1 bits and w_2 bits is of one clock cycle delay. For a given index i, we first shift w_1 bits by $\lfloor \frac{i}{w_1} \rfloor$ times and cache this position, then shift w_2 bits by $\lfloor \frac{i \bmod w_1}{w_2} \rfloor$ times, and finally shift the remaining bit by bit. For the next index j behind i, we starting shifting from the position we have cached when we recover index i, and then repeat identical steps as we previously do. The experimental results has shown that the tuple $\langle 50, 10, 1\rangle$ outperforms others, with the minimum of $14,956$ clock counts. Figure 5 depicts some

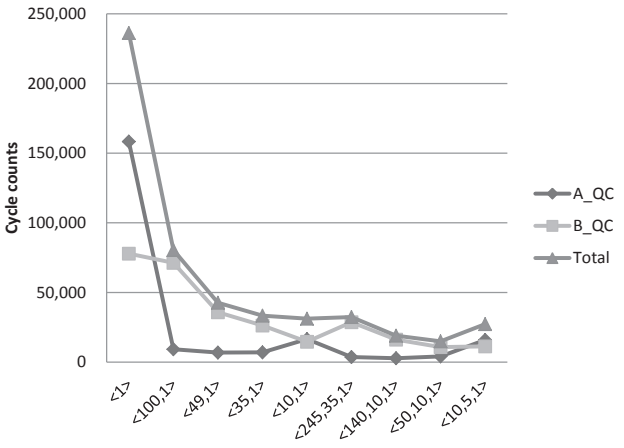


Fig. 5: The timing performance of different configurations of shift arrays for the proposed vector recovering unit by experiments.

typical values of $\langle w_1, w_2, 1\rangle$ and $\langle w_1, 1\rangle$ for comparisons. The task of extracting B_{QC} is much slower than that of A_{QC} since B_{QC} is denser. This also echoes that the total delay principally depends on the former one. Indeed, we can use a 4-level shifting structure to reduce more cycle counts but the improvement is trivial. Moreover, excessive complicated shifting arrays result in a very low operating frequency of the circuits. If users prefer a higher working frequency or even smaller slice utilizations, we suggest to use the 2-level $\langle 10, 1\rangle$ shifting because this configuration is also good, with about $31,236$ cycle counts.

4.2 Signature Verifier

The major operation that the signature verifier performs is the vector-matrix multiplication $H'_{QC} \cdot e'$, depicted in Figure 6. The public key H'_{QC} is a dense matrix and we cannot apply our memory efficient solution to the signature verifier. Consequently, we directly store the first row of each circulant blocks of H'_{QC} to the BRAM. To maintain a high memory access throughput, we use three block memories with 350-bit width, which allows us to extract 1050 bits per cycle. Thus it takes five clock cycles in total to obtain a complete row of H'_{QC}. Unlike sparse vectors we see in the signature generation, the public signature e' is dense, with weight equal to around 1602. We do not trace these 1602 indexes of e' to address rows of H'_{QC} because the number of indexes is too large and each index must be processed by an integer divider of 15 clock cycles delay as shown in Section 3.4. We do not have performance gain by indexing them and we therefore scan e' bit by bit using a 9800-bit cyclic shifter register: if the bit is '1', the associated row of H'_{QC} must be processed, otherwise it is discarded.

The processing of each non-zero bit is as following: The 10-bit address pointer increases by one whenever 50 bits of e' is scanned since the circulant block size p is 50. Then the 'base' row stored in BRAM is loaded and later quasi-cyclic shifted by $x \bmod 50$ assuming the current bit we scan is $e'[x]$. This QC shifted row is eventually summed to a 4900-bit register.

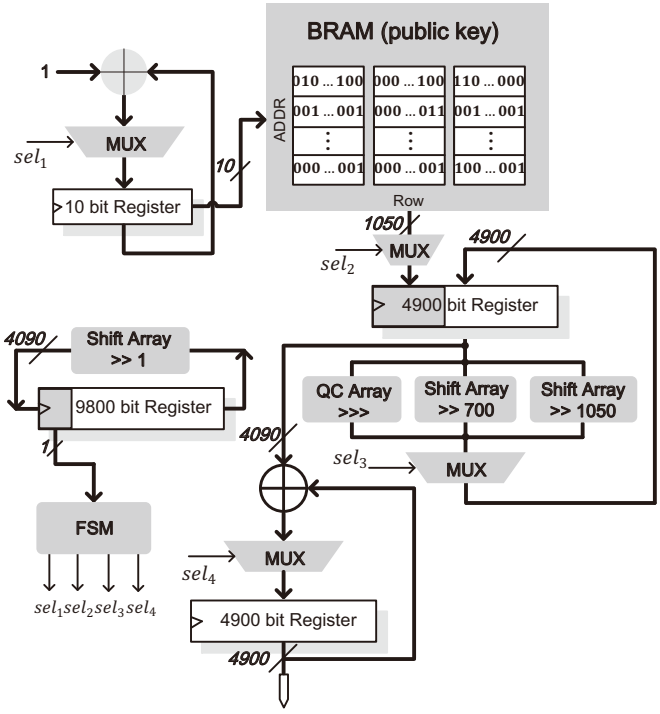


Fig. 6: General architecture of signature verifier for 80-bit security level

5 FPGA Implementation Results

In the following we present our QC-LDGM signature implementation results in FPGA. All our results are obtained post place-and-route for a Xilinx Virtex XC6VLX240T FPGA using Xilinx ISE 14.2. As shown in Table 5, our signature verifying engine runs at 260MHz and verifies a signature in 10,780 cycles, that is $41\mu s$ of operation time. Our first prototype of signature issuing engine runs at 220MHz but produces a signature very fast, taking approximately 2,218 cycles and is around a factor of 4.86 faster than verification. The significant reduction is primarily due to the different scanning patterns of these two operations: We must scan every bit of the signature to verify its validity whereas we only process those non-zero bits of s and u, which is much faster than verification. Our second prototype targeting at memory efficiency also achieves a competitive result, comparing with the first one. It runs at 150MHz and issues signatures in $176\mu s$ with almost doubled slice utilization. The slices increase due to the vector recovering unit, in which we have three large shifting arrays and two 9800-bit registers. Nevertheless, the memory utilization is much improved with 18 vs 60 RAM36E1 memory blocks due to the second memory storage strategy we propose.

Table 6 reports our results in the context of other known public-key signature scheme implementations, including code-based signatures [11], [18], [19], [20], lattice-based signatures [26], [27], [28], ECC [29] and RSA [30]. To the best of authors' knowledge, our results achieve the best speed record of code based signature for the time being. The previous fastest implementation is reported in [20] running on a single core of Intel i5 processor. Their fast software implementation exploits fast yet secure Goppa code decoding techniques which relies on an additive FFT

TABLE 5: Implementation results of our QC-LDGM signature with parameter $n = 9800, k = 4900, p = 50, w = 18$ on a Xilinx Virtex-6 XC6VLX240T FPGA after PAR. The clock count given here is an average value of 10,000 randomly generated signatures.

Aspect	Verifier	Generator (fast)	Generator (compact)
FFs	12,770 (4%)	22,803 (7%)	40,398 (13%)
LUTs	12,844 (8%)	23,587 (15%)	42,236 (28%)
Slices	4,153 (11%)	7,126 (18%)	15,180 (40%)
36Kb-BRAM	30 (7%)	60 (14%)	18 (4%)
Frequency	260 MHz	220 MHz	150 MHz
Time/Op	$41\mu s$	$10\mu s$	$176\mu s$
Verify Signature	10,780 cycles	—	—
Orthogonalize s	—	1,204 cycles	1,204 cycles
Compute e_1	—	1,652 cycles	712 cycles
Compute e_2	—	566 cycles	25,645 cycles
Overall Average	10,780 cycles	2,218 cycles	26,357 cycles

TABLE 6: Performance comparsion of our QC-LDGM FPGA implementations with other public-key signature schemes.

Scheme	SL (bit)	Platform	f[MHz]	Key Size	Time/Op	Cycles	FFs	LUTs	Slices	BRAM
This work(sign, fast)	80	Xilinx Virtex-6	220	1.8 Mb	$10\mu s$	≈2,218	22,803	23,587	7,126	60
This work(sign, compact)			150	253Kb	$176\mu s$	≈26,357	40,398	42,236	15,180	18
This work(ver)			260	938 Kb	$41\mu s$	≈10,780	12,770	12,844	4,153	30
McBits (sign) [20]	80	Intel Core i5	2500	20 Mb	$0.156s$	3.9×10^8	—	—	—	—
McBits (ver) [20]				160Mb	$0.87\mu s$	2,176	—	—	—	—
Parallel-CFS (sign) [11], [19]	80	Intel Xeon	3,200	20Mb	3.75s	—	—	—	—	—
CFS (sign) [18]	50	Xilinx XCV300E	62	1Mb	0.86s	5.4×10^7	—	—	2,593	18
Bimodal Lattice (sign) [26]	128	Xilinx Spartan-6	129	2Kb	$126\mu s$	$\approx 16,210$	7,033	7,491	2,431	7.5
Bimodal Lattice (ver) [26]			142	7Kb	$70\mu s$	9,835	4,488	5,275	1,727	4.5
Lyubash Lattice (sign) [27]	100	Intel Core i5	2500	2Kb	—	634,988	—	—	—	—
Lyubash Lattice (ver) [27]				12Kb	—	45,036	—	—	—	—
Lyubash Lattice (sign) [28]	100	Xilinx Virtex-6	204	2Kb	$79\mu s$	—	95,511	67,027	19,896	234
Lyubash Lattice (ver) [28]			156	12Kb	$69\mu s$	—	57,903	61,360	18,998	120
ECDSA (sign) [29]	128	Xilinx Virtex-5	20	—	7.15ms	$\approx 143,000$	32,299 LUT/FF pairs		—	—
ECDSA (ver) [29]				—	9.09ms	$\approx 181,800$	32,299 LUT/FF pairs		—	—
RSA (sign) [30]	80	Xilinx Virtex-5	450	1Kb	1.52ms	$\approx 684,000$	—	—	3,237	5

for fast root computation, a transposed additive FFT for fast syndrome computation, and a sorting network to avoid cache-timing attacks. It is reported in their paper that this software signs in than $0.425 \cdot 10^9$ Ivy Bridge cycles on average; the median is $0.391 \cdot 10^9$ Ivy Bridge cycles, which we roughly estimate to be 0.156 second per signature. Comparatively, our implementation produces a signature of the same security level in $10\mu s$, renovating this record with a factor of 15600. When comparing with another popular quantum resisted counterpart — lattice signatures, our implementation of code based signature is, for the first time, outperforms at timing performance, with slice and BRAM utilizations in almost same order of magnitude. For instance, the latest version of Bimodal Lattice FPGA implementation, published in 2014, generates a signature at $126\mu s$ occupying 2431 slices and 7.5 BRAMs. Our fast version of LDGM signature can run even faster at $10\mu s$ with 7126 slices and 60 BRAMs. Finally we also include some implementations of the standard signature algorithms such as RSA, ECDSA to have some insights of the potential of the code based signatures. In terms of the signing speed, the code based signature apparently runs much faster in magnitude; the circuit size of our code based signature is also approaching theirs. Even for the bottleneck — key size, of all code based signature schemes, our compact version of signature signing engine consumes 18 block RAMs. In contrast, RSA coprocessor in [30] uses 5. Our implementations reinforce the arguments that code based signature scheme is indeed a promising candidate for replacing the currently used RSA or ECC in the coming era of quantum computing.

6 CONCLUSION

In this paper, we implemented QC-LDGM code signature on smaller embedded systems, enabling the breaking of previous speed record of CFS signature implementations.

We proposed a simplified signature signing algorithm with faster speed and smaller secret key size. In the meantime, We also proposed a variant of constant weight coding, allowing for less memory overhead and very high coding efficiency close to 1. We addressed the most concerned secret key storage problem by proposed two memory management solutions. The first one enables a fast implementation, issuing one signature in 10 μs. The other compact one is memory efficient, by using only 18 memory blocks that can issue one signature in only 176 μs.

REFERENCES

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM journal on computing*, vol. 26, no. 5, pp. 1484–1509, 1997.

[2] L. M. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang, "Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance," *Nature*, vol. 414, no. 6866, pp. 883–887, 2001.

[3] N. Xu, J. Zhu, D. Lu, X. Zhou, X. Peng, and J. Du, "Quantum factorization of 143 on a dipolar-coupling nmr system," *arXiv preprint arXiv:1111.3726*, 2011.

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

[5] N. T. Courtois, M. Finiasz, and N. Sendrier, "How to achieve a Mceliece-based digital signature scheme," in *Advances in Cryptology-ASIACRYPT 2001*. Springer, 2001, pp. 157–174.

[6] G. Kabatianskii, E. Krouk, and B. Smeets, "A digital signature scheme based on random error-correcting codes," in *Crytography and Coding*. Springer, 1997, pp. 161–167.

[7] G. Kabatiansky, E. Krouk, and S. Semenov, *Error Correcting Coding and Security for Data Networks: Analysis of the Superchannel Concept*. John Wiley & Sons, 2005.

[8] P. S. Barreto, R. Misoczki, and M. A. Simplicio Jr, "One-time signature scheme from syndrome decoding over generic error-correcting codes," *Journal of Systems and Software*, vol. 84, no. 2, pp. 198–204, 2011.

[9] A. Otmani and J.-P. Tillich, "An efficient attack on all concrete kks proposals," in *Post-quantum cryptography*. Springer, 2011, pp. 98–116.

[10] H. Niederreiter, "Knapsack-type cryptosystems and algebraic coding theory," *Pproblems Of Control and Information Theory-Problemy Upravleniya I Thorii Informatsii*, vol. 15, no. 2, pp. 159–166, 1986.

[11] G. Landais and N. Sendrier, "CFS Software Implementation." *IACR Cryptology ePrint Archive*, vol. 2012, p. 132, 2012.

[12] M. Finiasz, "Parallel-CFS," in *Selected areas in cryptography*. Springer, 2011, pp. 159–170.

[13] M. Baldi, M. Bianchi, F. Chiaraluce, J. Rosenthal, and D. Schipani, "Using ldgm codes and sparse syndromes to achieve digital signatures," in *Post-quantum cryptography*. Springer, 2013, pp. 1–15.

[14] M. Baldi and F. Chiaraluce, "Cryptanalysis of a new instance of mceliece cryptosystem based on qc-ldpc codes," in *Information Theory, 2007. ISIT 2007. IEEE International Symposium on*. IEEE, 2007, pp. 2591–2595.

[15] M. Baldi, F. Chiaraluce, R. Garello, and F. Mininni, "Quasi-cyclic low-density parity-check codes in the mceliece cryptosystem," in *Communications, 2007. ICC'07. IEEE International Conference on*. IEEE, 2007, pp. 951–956.

[16] M. Baldi, M. Bodrato, and F. Chiaraluce, "A new analysis of the mceliece cryptosystem based on qc-ldpc codes," in *Security and Cryptography for Networks*. Springer, 2008, pp. 246–262.

[17] M. Baldi, M. Bianchi, F. Chiaraluce, J. Rosenthal, and D. Schipani, "Enhanced public key security for the mceliece cryptosystem," *Journal of Cryptology*, vol. 29, no. 1, pp. 1–27, 2016.

[18] J.-L. Beuchat, N. Sendrier, A. Tisserand, and G. Villard, "Fpga implementation of a recently published signature scheme," 2004.

[19] G. Landais and N. Sendrier, "Implementing cfs," in *Progress in Cryptology-INDOCRYPT 2012*. Springer, 2012, pp. 474–488.

[20] D. J. Bernstein, T. Chou, and P. Schwabe, "Mcbits: fast constant-time code-based cryptography," in *Cryptographic Hardware and Embedded Systems-CHES 2013*. Springer, 2013, pp. 250–272.

[21] N. Sendrier, "Encoding information into constant weight words," in *Information Theory, 2005. ISIT 2005. Proceedings. International Symposium on*. IEEE, 2005, pp. 435–438.

[22] T. M. Cover, "Enumerative source encoding," *Information Theory, IEEE Transactions on*, vol. 19, no. 1, pp. 73–77, 1973.

[23] G. Goloumb, "Run length encoding," *Information Theory, IEEE Transactions on*, vol. 12, pp. 399–401, 1966.

[24] S. Heyse, "Low-reiter: Niederreiter encryption scheme for embedded microcontrollers," in *Post-Quantum Cryptography*. Springer, 2010, pp. 165–181.

[25] S. Heyse and T. Güneysu, "Towards one cycle per bit asymmetric encryption: code-based cryptography on reconfigurable hardware," in *Cryptographic Hardware and Embedded Systems–CHES 2012*. Springer, 2012, pp. 340–355.

[26] T. Pöppelmann, L. Ducas, and T. Güneysu, "Enhanced lattice-based signatures on reconfigurable hardware," in *Cryptographic Hardware and Embedded Systems–CHES 2014*. Springer, 2014, pp. 353–370.

[27] T. Güneysu, T. Oder, T. Pöppelmann, and P. Schwabe, "Software speed records for lattice-based signatures," in *Post-Quantum Cryptography*. Springer, 2013, pp. 67–82.

[28] T. Güneysu, V. Lyubashevsky, and T. Pöppelmann, "Practical lattice-based cryptography: A signature scheme for embedded systems," in *Cryptographic Hardware and Embedded Systems–CHES 2012*. Springer, 2012, pp. 530–547.

[29] B. Glas, O. Sander, V. Stuckert, K. D. Müller-Glaser, and J. Becker, "Prime field ecdsa signature processing for reconfigurable embedded systems," *International Journal of Reconfigurable Computing*, vol. 2011, p. 5, 2011.

[30] D. Suzuki and T. Matsumoto, "How to maximize the potential of FPGA-based DSPs for modular exponentiation," *IEICE transactions on fundamentals of electronics, communications and computer sciences*, vol. 94, no. 1, pp. 211–222, 2011.