An Application Specific Instruction Set Processor (ASIP) for the Niederreiter Cryptosystem

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

***Abstract*—The Niederreiter public-key cryptosystem is based on the security assumption that decoding generic linear binary codes is NP complete, and therefore, is regarded as an alternative post-quantum solution to resist quantum computing. Current hardware implementations for the Niederreiter cryptosystem focus on data encryption/decryption but few of them consider digital signature producing given that signature scheme is much different from encrytion/decrytion and complicated to be integrated. In this work, we address the problem of achieving efficient Niederreiter digital signature and extending it to execute encryption/decryption on reconfigurable hardware. We first present a new parameter selection method by which both encryption/decryption and signature are able to be performed with the same hardware configurations. Then we design a compact ASIP architecture with the proposed parameter selection and resource sharing elaboration. FGPA experiments show that the proposed unified architecture can achieve encryption, decryption and signature with $1.41\ \mu s$, $798.57\ \mu s$ and $14.07\ s$ respectively while maintaining acceptable area tradeoffs ($4254\times$slices, $29\times$36Kb-BRAMs and $3\times$DSP48E1s) on Virtex-6 devices.**

***Index Terms*—Cryptographic hardware and implementation, Application specific instruction set processor, Niederreiter cryptosystem, FPGA.**

I. Introduction

MOST currently popular public-key cryptographic systems rely on the integer factorization problem or discrete logarithm problem, both of which would be easily solvable on large enough quantum computers using Shor's algorithm [31]. Even though current publicly known quantum computing is not powerful enough to attack real cryptographic systems, many cryptographers are researching new algorithms in case quantum computing becomes a threat in the future. This work is referred now as post-quantum cryptography [1]. One of the potential candidate for the post-quantum cryptography is the code-based cryptography based on the hardness of decoding a general linear code. The code-based scheme is categorized into the McEliece cryptosystem [21] and its variant the Niederreiter cryptosystem [22]. The Niederreiter cryptosystem has two advantages against the McEliece cryptosystem: 1) It has a smaller public key size for the same security and thus it is more efficient to be employed on small and embedded systems. 2) The Niederreiter's proposal enables a practical digital signature scheme but the McEliece does not. Based on these two merits, we stay focus on the implementation of the Niederreiter cryptosystem in this paper and hope to establish the necessary confidence for code-based cryptosystem deployment in real-world applications.

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.

To the best of our knowledge, there exist three publicly available hardware implementations of the Niederreiter cryptosystem, targeting exclusively either at decryption/encryption or signature. The first one is an implementation for small 8-bit AVR microcontrollers, which reports an encryption time of $1.6\ ms$ and a decryption time of $179\ ms$ [10]. The most recent Niederreiter implementation is carried out on reconfigurable hardware, enabling encryption and decryption in $0.66\ \mu s$ and $58.78\ \mu s$ [11] on a Xilinx Virtex-6 FPGA. The only hardware implementation that we are aware of for the Niederreiter signature scheme [17] reports an average signing time of 0.86 seconds. Nevertheless, they do not implement a full version of the signature scheme but two main steps of it. Furthermore, their implementation for the original signature scheme cannot resist DOOM-GBA attack [8]. Our work first of all aims at providing the first hardware prototype for digital signatures and then further extending it to be the first solution attempting to integrated both data encryption and digital signature for reconfigurable hardware.

Our contribution: 1) In this work, we propose an application specific instruction set processor for the Niederreiter cryptosystem using binary Goppa code. We particularly focus on evaluations and improvements of the signature issuing and its combination of data encryption/decryption on a unified architecture. 2) We show that by choosing an appropriate parameter set, the proposed cryptoprocessor can provide desirable efficiency in terms of computational complexity for encryption ($1.41\ \mu s$), decryption ($798.57\ \mu s$), signature ($14.07\ s$) and verification ($1.84\ \mu s$) on Xilinx Vertex-6 FPGAs, respectively. 3) We also show that it is possible to implement the complete Niederreiter cryptosystem with acceptable area/time tradeoff on reconfigurable hardware. The source code of this project is available under https://github.com/davidhoo1988/NiederreiterCryptoprocessor.

Our paper is organized as follows. We first introduce the Niederreiter cryptosystem and how we select the system parameters for implementation in Section II. In Section III, we describe in detail about our unified processor architecture for encryption/decryption and signature/verification. Section IV presents our implementation results on a Xilinx Virtex-6 FPGA. Finally, Section V concludes our work.

II. Goppa Code and Niederreiter Cryptosystem

The Niederreiter cryptosystem is a variation of the McEliece cryptosystem developed in 1986 by Harald Niederreiter [22]. The Niederreiter cryptosystem uses a syndrome as ciphertext

and the message as an error pattern (The hamming weight of this pattern is restricted to a certain number). It applies the same idea to the parity check matrix H of a linear code as the McEliece cryptosystem does, but obtains much smaller public key size. Another prominence Niederreiter has against the McEliece Cryptosystem is that it can be used to construct a digital signature scheme — CFS signature [6].

A. Goppa Code

In this subsection, we briefly introduce the binary Goppa code that the McEliece/Niederreiter cryptosystem uses as the underlying linear code for data encryption, decryption and signature. The Goppa code $\Gamma(L, g(z))$ is defined by the Goppa polynomial $g(z)$ which is a polynomial of degree t over the extension field $GF(2^m)$, and a support L of $GF(2^m)$.

***Definition** 2.1:*

$$\Gamma(L, g(z)) = (c_0, \ldots, c_{n-1}) \in \{0,1\}^n, n = 2^m$$

where $\sum_{j=0}^{n-1} \frac{c_j}{z-a_j} \bmod g(z) = 0$, $g(z) = \sum_{i=0}^{t} g_i z^t$ and $L = \{\alpha_0, \ldots, \alpha_{n-1}\} \subseteq GF(2^m)$.

Let $c' = (c'_0, \ldots, c'_{n-1})$ be a received word, containing r errors, with $r \leq t$ (t is the degree of the polynomial $g(z)$ in Goppa code $\Gamma(L, g(z))$) as follows:

$$(c'_0, \ldots, c'_{n-1}) = (c_0, \ldots, c_{n-1}) + (e_0, \ldots, e_{n-1})$$

with $e_i \neq 0$ in exactly r or less than r places. To correct the word, and find the right codeword $c = (c_0, \ldots, c_{n-1})$, we have to find the error vector $e = (e_0, \ldots, e_{n-1})$ and therefore to determine the set of error locations $B = \{i | e_i \neq 0\}$. In coding theory, the error vector e can be obtained via error locator polynomial $\sigma(z)$ from syndrome decoding (Algorithm 1) where the syndrome $s(z)$ and error locator $\sigma(z)$ are defined as shown in Definition 2.2:

***Definition** 2.2:*

$$s(z) = \sum_{i=0}^{n-1} \frac{c'_i}{z - \alpha_i} = \sum_{i=0}^{n-1} \frac{e_i}{z - \alpha_i} \bmod g(z)$$

$$\sigma(z) = \prod_{i \in B} (z - \alpha_i), \alpha_i \in L$$

After the error locator polynomial $\sigma(z)$ is obtained, the last remaining steps are to search the roots of $\sigma(z)$ i.e. $\{\alpha_i | \sigma(\alpha_i) = 0 \bigcap \alpha \in L\}$ such that the error vector e can be determined by the indices i of α_i. This is, in fact, the most time-consuming operation in decoding procedure [17].

B. Data Encryption and Decryption

The relation between the Niederreiter cryptosystem and Goppa code is illustrated as follows: a coding system generates the corresponding generator matrix and parity check matrix to encode or decode a message. Likewise, Niederreiter scheme exploits this coding system to encrypt the plaintext by coding it and to decrypt the ciphertext by decoding it. An essential aspect of the Niederreiter cryptosystem relates to the selection of the coding system, as the security of the cryptosystem relies on the difficulty of how to decode the coding system without known a particular set of parameters generated by the system. In particular, the Niederreiter system is secure when used with a binary Goppa code and thus it is adopted as the coding system for the Niedereiter scheme.

Algorithm 1: Goppa code decoding using Patterson's Algorithm [23]

Input: Syndrome polynomial $s(z)$, irreducible Goppa polynomial $g(z)$ and Goppa code support L
Output: Error locator polynomial $\sigma(z)$

1 Use the extended Euclidean algorithm to find $T(z)$ such that $s(z)T(z) \equiv 1 \bmod g(z)$
2 **if** $T(z) = z$ **then**
3 $\quad \sigma(z) = z$
4 **else**
5 $\quad$ Calculate $d(z) = \sqrt{T(z) + z} \bmod g(z)$
6 $\quad$ Use the extended Euclidean algorithm to find $a(z)$ and $b(z)$ with $b(z)$ of least degree, satisfying $d(z)b(z) \equiv a(z) \bmod g(z)$
7 $\quad \sigma(z) = a^2(z) + zb^2(z)$
8 **return** $\sigma(z)$

Encryption and Decryption are described as follows: Suppose Bob wishes to send a message m to Alice whose public key is $\{\hat{H}, t\}$, where $\hat{H} = SHP$, S is the randomly selected non-singular matrix, P is the randomly selected permutation matrix, H is the parity check matrix of Goppa code $\Gamma(L, g(z))$ and t is the degree of polynomial $g(z)$. Bob uses Algorithm 2 to encrypt the message m.

Algorithm 2: Niederreiter Message Encryption

Input: message vector m, public key $pk = \{\hat{H}, t\}$
Output: ciphertext c

1 Bob encodes the message m as a binary matrix/vector of length n and weight at most t.
2 Bob computes the ciphertext as $c = \hat{H}m^T$, m^T is the transpose of matrix m.
3 **return** c

Upon receipt of $c = \hat{H}m^T$ from Bob, Alice performs Algorithm 3 to retrieve the message m with her private key $\{S, P, g(z), L\}$. Typically, matrix inverses in step 1 are pre-computed and restored for speed-up.

C. CFS Digital Signature Scheme

The signature scheme is shown as following:

1) Hash the document $d \rightarrow h_d$ (with a public hash algorithm).
2) Decrypt h_d as if it were an instance of ciphertext ($h_d \rightarrow d'$) using the private key.
3) Append the decrypted message d' to the document d as a signature.

Signature verification applies the public encryption function to the signature and checks whether or not this equals the hash value of the document. However, when using Niederreiter, or

Algorithm 3: Niederreiter Message Decryption

Input: ciphertext c, secret key $sk = \{S, P, g(z), L\}$
Output: recovered message m

Alice computes the inverse of P and S (i.e. P^{-1}, S^{-1}).
Alice computes $S^{-1}c = HPm^T$ to get the syndrome polynomial $s(z)$.
Alice applies Algorithm 1 with inputs $s(z)$, $g(z)$ and L to obtain the error locator polynomial $\sigma(z)$.
Alice searches all roots α_i of $\sigma(z)$ to determine $B = \{i|\sigma(\alpha_i) = 0\}$
Alice recovers the error vector $e = (e_0, \ldots, e_{n-1}) = Pm^T$ where $e_i = 1$ for $i \in B$ and $e_i = 0$ elsewhere.
Alice computes the message m via $m^T = P^{-1}e$.
return m

in fact any cryptosystem based on error correcting codes, the second step of the signature scheme almost always fails and the probability of this failure is referred as failure rate of the signature. The failure rate exists because a random syndrome usually corresponds to an error pattern of weight greater than t. In other words, it is difficult to generate a random ciphertext for issuing signatures without failure unless it is explicitly produced as an output of the encryption algorithm.

In 2001, Courtois, Finiasz and Sendrier proposed to use almost complete decoding to tackle this problem [6] (CFS signature scheme). The idea is that, for example, assume the hashed document h_d corresponds to the error pattern with $t + \delta$ errors, then repeat the following procedure: randomly reverse the values in δ positions of the error pattern followed by decoding it. Within finite iterations h_d would becomes decodable. This is because at the end of the iteration, the randomly selected δ positions are all corrected and thus the left t positions of errors would be easily found out via the efficient syndrome decoding algorithm. To successfully produce a valid signature, [6] also estimates the average number of decoding is close to $t!$ and the total signature time would be significantly reduced if we could simplify the root finding of $\sigma(z)$ as mentioned in Section II A. One solution is to perform the divisibility test prior to the root finding, that is, to check whether $z^{2^m} \equiv z \bmod \sigma(z)$ [6] and therefore we only have to compute the roots for once during the signature producing. The detailed flow of CFS signature is shown in Algorithm 4.

D. Parameter Selection

The primary target of our implementation for the Niederreiter cryptosystem is to apply an identical set of parameters for both encryption/decryption and signature. Previous work for the hardware implementation of the Niederreiter cryptosystem [10], [11] aim at optimizing the timing performance of encryption/decryption with secure parameters. But in our cases, we consider both encryption/decryption and signature/verification for an unbiased evaluation. The basic principal of our parameter selection is to maximize the speed performance with acceptable memory overhead in the context of roughly 80-bit security level.

Algorithm 4: CFS Signature Scheme

Input: document hash h_d, secret key $sk = \{S, P, g(z), L\}$, public key $pk = \{\hat{H}\}$
Output: document hash d with signature d'

while *True* **do**
    Generate δ random integer numbers $\{r_1, r_2, \ldots, r_\delta\}$.
    Re-calculate syndrome polynomial $s(z)$ according to random numbers R, document hash h_d and public key $\hat{H}$.
    Decode syndrome $s(z)$ to get the error locator $\sigma(z)$ using Algorithm 1 with the private key sk.
    if $z^{2^m} \equiv z \bmod \sigma(z)$ **then**
        Find all roots α_i of $\sigma(z)$ to determine $B = \{i|\sigma(\alpha_i) = 0\}$
        Set $d' = (d_0, \ldots, d_{n-1})$ where $d_i = 1$ for $i \in B \bigcup R$ and $d_i = 0$ elsewhere.
        break
return $\{d,d'\}$

The fastest known attacks against the Niederreiter cryptosystem is the information set decoding (ISD) [33] and for safety use of this system, we must select security parameters against ISD and its variants [2], [9], [20]. Table I lists commonly used security parameters for the Niederreiter cryptosystem. According to this table, we cannot use the parameters proposed by Bernstein et al. [2] originally intended for encryption. This is primarily because with these parameters, the failure rate of signature is almost 100%. Instead, we select the parameters in the remaining parameter space proposed by Finiasz et al. [8] such that the failure rate for CFS signature is approximately 0. The major problems for these parameters are the very long run time and the significant large public key size. For instance, $n = 2^{20}, m = 20, t = 8$ offers the shortest signature time but 20 Mbits of the public key size is too large to be supported by any type of Xilinx FPGAs. Therefore, we select the parameter set $n = 2^{16}, m = 16, t = 9$ for both encryption and signature after a consideration of the security level (76.5-bit security), running time and memory size.

ISD algorithms are particularly efficient for solving instances of the syndrome decoding problem which have a single (or a few) solutions. For CFS signature scheme, there are very likely a large number of solutions and a more threatening attack is the generalized birthday algorithm (GBA) attack [35]. Finiasz et al. [8] proposed another countermeasure, called parallel-CFS, against such attack for improving the security of the CFS signature scheme. The idea of this method is to produce λ (typically two to four) hashes from the document to be signed and to sign each of those hash values separately. The final signature will be the collection of all those hash values [17]. The parameter λ is highly related to the security level of the CFS signature scheme. Table II lists the security levels of the parameter set we have selected for hardware implementation against such new attack. We use the parameter λ from this table to implement the actual parallel-CFS signature on our proposed hardware. Readers can also verify that more

TABLE I
SOME PARAMETER ANALYSIS FOR THE NIEDERREITER CRYPTOSYSTEM AGAINST INFORMATION SET DECODING (ISD) ATTACKS

(n, m, t)	ISD Security*	Failure Rate†	Run Time⋆	Public/Secret Key Size (Mbits)	Supported Xilinx FPGAs	Application
$(2^{10}, 10, 38)$	2^{60}	≈ 1	—	$0.371/0.148$	Virtex-II, $4, 5, 6, 7$	Encryption
$(2^{11}, 11, 27)$	2^{80}	≈ 1	—	$0.580/0.106$	Virtex-II, $4, 5, 6, 7$	Encryption
$(2^{16}, 16, 9)$	$2^{76.5}$	2^{-155}	5.995×10^8	$9/1.019$	Virtex-$5, 6, 7$	Signature
$(2^{18}, 18, 9)$	$2^{84.5}$	2^{-2483}	6.306×10^8	$40.5/4.525$	Virtex-7	Signature
$(2^{20}, 20, 8)$	2^{81}	$2^{-437111}$	6.258×10^7	$160/20.025$	—	Signature

*The security level is measured by the number of binary operations. The first two sets of parameters are evaluated by Bernstein et al. [2] for encryption and the last three sets by Finiasz et al. [8] for signature.

†We use the methods proposed by [8] to estimate these values.

⋆The run time formula is explained in Appendix A.

numbers of the parallel signatures to perform indicate a better GBA security but do not affect ISD security. On the other hand, the run time of parallel-CFS increases by a factor of λ.

TABLE II
THE SECURITY PARAMETERS THAT WE HAVE CHOSEN FOR THE PARALLEL-CFS AGAINST BEST KNOWN ATTACKS INCLUDING GBA AND ISA ATTACKS

(n, m, t)	λ^*	ISD Security	GBA Security	Run Time	Failure Rate
$(2^{16}, 16, 9)$	1	$2^{76.5}$	$2^{53.6}$	5.995×10^8	2^{-155}
	2		$2^{68.7}$	1.199×10^9	2^{-154}
	3		$2^{74.9}$	1.799×10^9	2^{-153}

*λ denotes the number of parallel signatures to perform. If $\lambda = 1$, it is exactly the original CFS signature scheme. $\lambda > 3$ can never compensate the cost in signature time and size of the signature. See [17] for more details.

III. APPLICATION SPECIFIC INSTRUCTION SET PROCESSOR FOR THE NIEDERREITER CRYPTOSYSTEM

We introduce in this section the internal structure of the proposed cryptoprocessor, shown in Fig. 1. There are five main structure components:

- Control unit (instruction fetching unit and instruction decoding unit): Controls the operation of the processor.
- Arithmetic and logic unit (ALU and PRNG): Perform data processing functions.
- Registers (GPRFs and SPRFs): Provides storage internal to the processor.
- Main memory: Stores instructions and data.
- System bus: Some mechanism that provides for communication among the control unit, ALU, and registers.

User can write their own programs e.g. encryption, decryption, signature, verification in the form of assembly language with the instruction set we have designed. Then a simplified assembler written in Perl translates user's program into machine binaries. These binaries are eventually uploaded into the instruction RAM for our cryptoprocessor to process. To fully exploit the main memory storage, user is only allowed to upload one application program (encryption, decryption, signature, verification) and its corresponding data at a time into instruction RAM and data RAM respectively. For instance, user should upload the encryption binaries to the instruction RAM and the public key and the plaintext to the data RAM when encrypting messages whereas he should switch to decryption the binaries, the secret key and the ciphertext when decrypting a message.

A. Instruction Set

With a thorough analysis of the algorithms used in the Niederreiter cryptosystem, we design in total 22 instructions listed in Table VII in the form of assembly codes. These instructions are roughly divided into 4 categories: data transfer, arithmetic & logic,random number generation and transfer of control. Readers can refer to Appendix B for more details of these instructions.

B. Instruction Fetching Unit and Decoding Unit

The instruction fetching unit is designed to fetch instructions from instruction memory according to the execution of assembly code programmers have written. Normally, instructions are executed sequentially and this can be done by increasing the program counter register by one. In the meantime, a large number of instruction branches exist in the Niederreiter cryptosystem, and therefore the conditional and unconditional jump instructions are implemented to fulfil this requirement.

Instruction decoding unit is used to decode the instructions obatined from the instruction fetching unit. The input instructions after decoding are then decomposed to a bunch of useful signals sent to other components including instruction fetching unit, memory, GPRF, SPRF, ALU and PRNG.

C. GPRF and SPRF

A register file is an array of processor registers and used to store data between memory and the function units (e.g. ALU, PRNG). In our design, the register file is refined to two categories: general general purpose register file (GPRF) and special purpose register file (SPRF). GPRFs are used for internal storage of the computational results whereas SPRFs are particularly designed for loop control and memory addressing. The detailed differences of these two register files can be found in Appendix C.

D. ALU structure

ALU is that part of the proposed ASIP that actually performs arithmetic and logical operations on data. All of the other

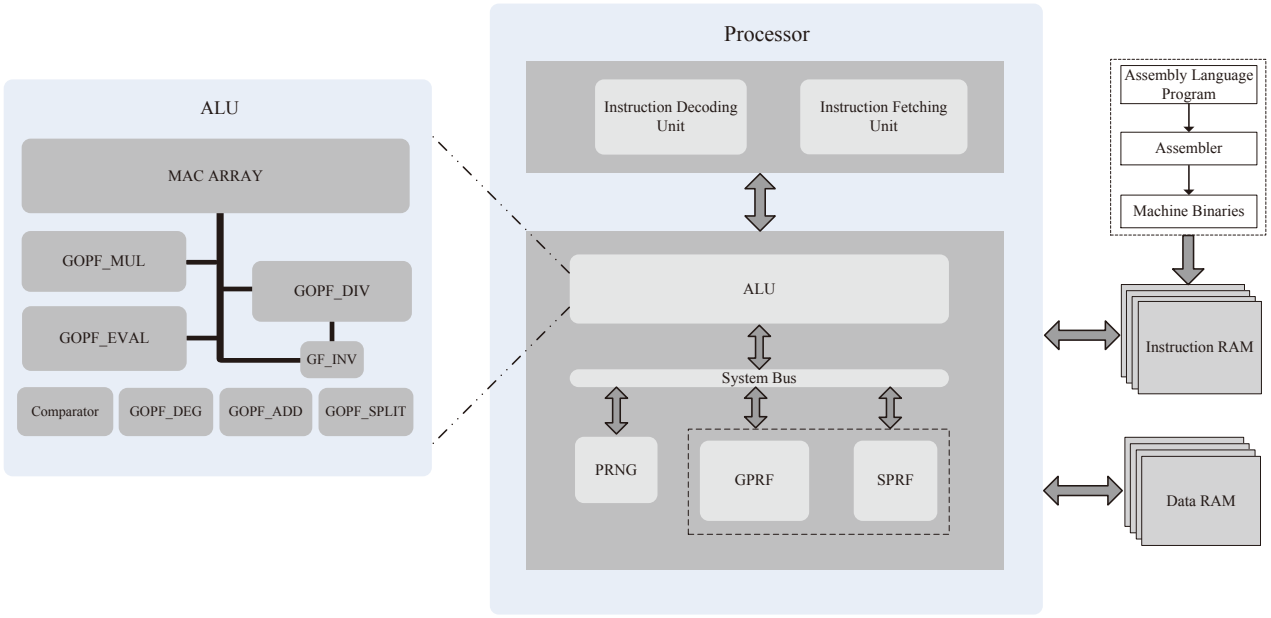

Fig. 1. System architecture for the proposed Niederreiter cryptoprocessor which includes five main components: Control unit (instruction fetching unit and instruction decoding unit), arithmetic and logic unit (ALU and PRNG), registers (GPRFs and SPRFs), main memory (instruction RAM and data RAM) and system bus. The PRNG is exclusively required by the CFS signature scheme for producing random integer numbers. The ALU is the most complex component which handles all arithmetic computation, for example, finite field multiplication & inversion, polynomial multiplication & division, root finding and some parts of logical computation like unconditional jump. To improve the performance of the ALU, a delicate MAC array composed of 9 multiplier-accumulation operators is shared by other critical units including polynomial multiplier, divisor, evaluator and inverter.

elements of the processor — control unit, registers, memory, PRNG — are there mainly to bring data into the ALU for it to process and then to take the results back out. ALU, in a sense, is the essence of the processor. Table VII in Appendix B (Arithmetic and Logic) lists all computational operations for the Niederreiter cryptosystem that we have implemented in the ALU. In this section, we describe in very detail about how these key ALU operations are realized.

1) Multiply-accumulator (MAC): To perform the arithmetic operations, the most fundamental thing is to build up finite field multiplication and addition over $GF(2^{16})$. In our previous design, multiplication and addition are separated and thus it takes $GF(2^{16})$ multiplier one clock cycle to do multiplication and then another clock cycle to do addition using $GF(2^{16})$ adder. However, we observe that the most computational intensive operations, including polynomial multiplication, polynomial division and polynomial evaluation, share a similar multiply-and-accumulate operation ($a + b * c$, where a, b, c are all $GF(2^{16})$ elements). That is, multiplication of two $GF(2^{16})$ elements is right followed up by adding another $GF(2^{16})$ element. As $GF(2^{16})$ multiplication is rather small with 16 bits operand size and $GF(2^{16})$ addition is nothing but exclusive gate operation, to merge $GF(2^{16})$ addition into $GF(2^{16})$ multiplication has little negative effect on the critical path but improves the timing performance significantly. In this way, the new MAC is capable of executing one multiplication and one addition in a single clock cycle delay.

Fig. 2 depicts the detailed structure of the proposed MAC module. Note that the multiplier employed in MAC is a straightforward implementation of Mastrovito product matrix approach [18], [19]. In fact, a variety of timing or area efficient multipliers are proposed in literature [4], [24] but they appear to be efficient when the operand size is considerably large, for example, 512 bits or even larger. In our application, the

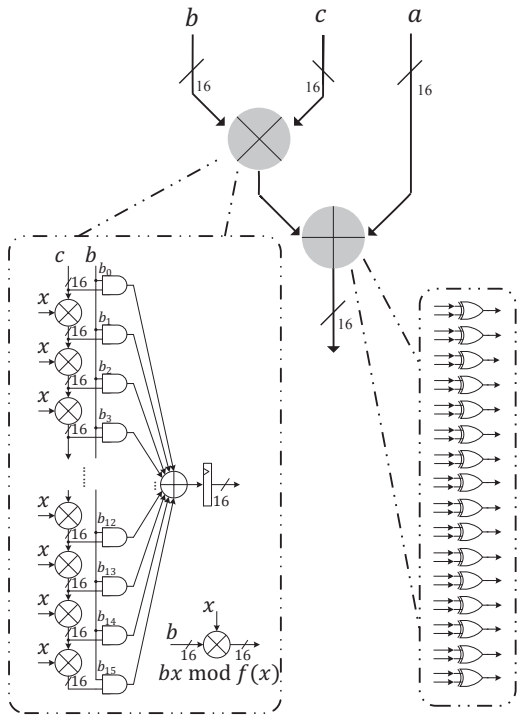

Fig. 2. Multiply-accumulate module (MAC) for performing $a + b * c$ over $GF(2^{16})$. The primitive polynomial used for this field is $f(x) = x^{16} + x^3 + x^2 + 1$.

TABLE III
COMPUTATIONAL STEPS FOR ITA INVERTER TO FIND INVERSE $b = a^{-1} \in GF(2^{16})$

Step	With $GF(2^{16})$ Multiplier	With $GF(2^{16})$ Squarer
	Initialize $b = a$	
1	—	$r = b^2$
2	$b = a \times r$	—
3	—	$r = b^2$
4	$b = a \times r$	—
5	—	$r = b^{2^3}$
6	$b = a \times r$	—
7	—	$r = b^2$
8	$b = a \times r$	—
9	—	$r = b^{2^7}$
10	$b = a \times r$	—
11	—	$r = b^2$
12	$b = a \times r$	—
13	—	$b = b^2$
	Return b	

operand size is as small as 16 bits in which the classic Mastrovito multiplier is more advantageous in terms of both timing and area performance.

2) Inverter: Multiplicative inverse of a finite field element is necessary in the Niederreiter cryptosystem from two aspects: First, one needs to compute inverse of the leading term of the divisor in polynomial division; Second, to normalize the error locator polynomial $\sigma(z)$, one also needs to calculate the inverse of the leading term of the $\sigma(z)$.

Basically there are two approaches to achieve multiplicative inverse over $GF(2^m)$: Euclidean extended algorithm (EEA) and Itoh and Tsujii algorithm (ITA) [14]. We have adopted ITA method for inverse because the existing MAC module can be directly exploited to do the inverse without complicated controls whereas EEA requires trial divisions and hence appears to be less attractive for implementation on hardware.

In our case, we just need to do inverse on $GF(2^{16})$ thus, all steps in ITA are deterministic, indicating it is beneficial for hardware implementation to unfold the loop inside the algorithm. In fact, a single multiplier and a squaring unit can achieve inverse. To illustrate this point, we list Table III the step-by-step procedure to calculate $GF(2^{16})$ inverse. It is easy to observe from this table that the multiplier and the squarer interleave executions to obtain the inverse. For the sake of better timing performance, three types of squaring units are implemented — b^2, b^{2^3}, b^{2^7} and therefore both squaring and multiplication can be done in one clock cycle respectively. Fig. 3 illustrates the ITA inversion architecture with a squaring block and a MAC. Note that the existing MAC module works here as a pure $GF(2^{16})$ multiplier with its addition operand set to zero.

3) Goppa Field Polynomial Multiplier : The polynomial multiplication in this work refers exactly to the multiplication operation for an extension field of $GF(2^{16})$, denoted as Goppa field $G \simeq GF(2^{16})[z]/g(z)$ where $g(z) = z^9 + g_8 z^8 + \ldots + g_0$ is the irreducible Goppa polynomial. Define two polynomials in $GF(2^{16})[z]/g(z)$ as $A(z) = a_0 + a_1 z + \ldots a_8 z^8$ and $B(z) = b_0 + b_1 z + \ldots + b_8 z^8$. Equation 1 represents an alternative way for Goppa field polynomial multiplication. From this equation, one can see that to do Goppa field polynomial multiplication, three steps are required — polynomial multiplying z ($z^i \cdot A(z)$), scalar multiplication ($b_i \cdot A(z)$) and accumulation. To multiply polynomials by z (Eq. (2)), first a right shift operation is performed to get $a_0 z + a_1 z^2 + \ldots + a_8 z^9$, then if the leading coefficient a_8 is non-zero, another scalar multiplication is performed to obtain $a_8 z^9 \bmod g(z)$, that is $a_8 z^9 = a_8 \cdot (g(z) + z^9) \bmod g(z)$. Finally, the partial product are all summed to get the result.

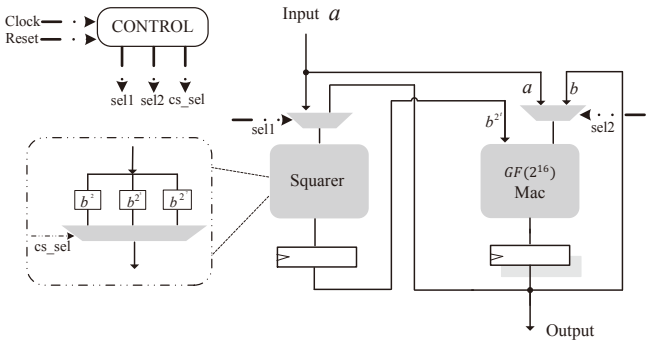

Fig. 3. $GF(2^{16})$ ITA inverter for computing $GF(2^{16})$ inverse.

$$\begin{aligned} A(z) \cdot z =& (a_0 + a_1 z + a_2 z^2 + \ldots + a_8 z^8) z \bmod g(z) \\ =& a_0 z + a_1 z^2 + \ldots + a_8 z^9 \bmod g(z) \\ =& a_0 z + a_1 z^2 + \ldots + a_8 \cdot (g(z) + z^9) \end{aligned} \quad (2)$$

At this stage, we can introduce the techniques we have used in Goppa field polynomial multiplier (GOPF_MUL). The first technique is that those aforementioned three steps share the same array of 9 MACs. Fig. 4 helps illustrate how it works. The MAC array enables the computation like $A(z)b_i + A'(z)$ where $A(z), A'(z)$ are $GF(2^m)[z]$ polynomials and b_i is a $GF(2^m)$ element, which combines the second step — scalar multiplication and the third step — accumulation together. On the other hand, the first step — polynomial multiplying z itself is also a combination of scalar multiplication and addition (See Eq. (2)), it is also natural to share exactly the same MAC array for the computation in which we set $A(z) = g(z) + z^9$, $A'(z) = a_0 z + \ldots + a_7 z^8$ and $b_i = a_8$.

The second technique to boost the performance is that the application of GOPF_DEG which computes the degree of polynomial $B(z)$. As the degree of the input multiplicand polynomial $B(z)$ is uncertain, ranging from 0 to 9, time delay is reduced by pre-computing the degree such that we do not have to traverse every coefficient b_i of $B(z)$ but terminate the computation as early as possible. The control logic of this multiplier shows the work flow: The multiplier is, first of all, reset to state PRE and then immediately runs into state DEGREE to obtain the degree of polynomial $B(z)$. After this, the multipliers starts the major three steps in Goppa field polynomial multiplication by commuting between state MAC and state SHIFT.

4) Goppa Field Polynomial Divider: A polynomial division is critical and needed in the extended Euclidean algorithm. Two things are considered in our implementation: First, both remainder and quotient must be obtained and therefore we

$$
\begin{aligned}
A(z) \cdot B(z) =& A(z)(b_0 + b_1 z + \ldots + b_{t-1} z^8) \bmod g(z) \\
=& A(z)b_0 + zA(z)b_1 + \ldots + z^8 A(z)b_8 \bmod g(z)
\end{aligned} \tag{1}
$$

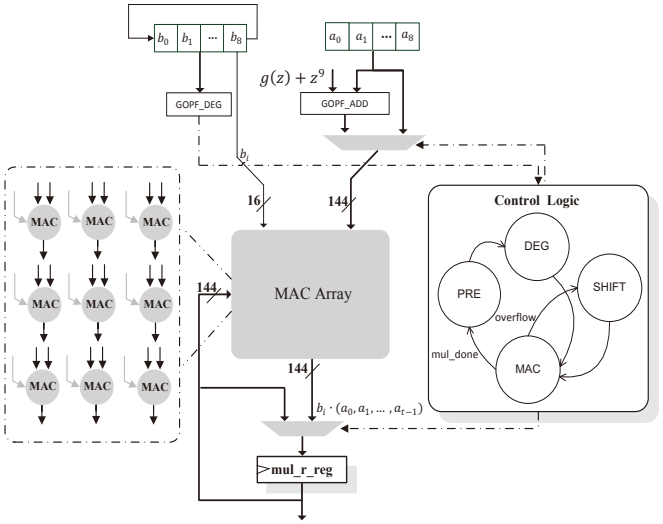

Fig. 4. Goppa field polynomial multiplier for computing multiplication over $GF(2^{16})[z]/g(z)$.

Algorithm 5: Control Flow of Polynomial Division $A(z) \div B(z)$

Input: dividend polynomial $A(z)$ and divisor polynomial $B(z)$
Output: quotient polynomial $Q(z)$ and remainder polynomial $R(z)$

1 Initialize quotient $Q(z) = 0$, remainder $R(z) = A(z)$, $deg(A(z)) = 0$ and $deg(B(z)) = 0$
2 **while** $deg(A(z)) \geq deg(B(z))$ **do**
3 Calculate polynomial degree $deg(A(z))$ and $deg(B(z))$
4 $P(z) =$ **ldcoeff**$(B(z))^{-1} \cdot$ **ldcoeff**$(A(z)) \cdot z^{deg(A(z)) - deg(B(z))}$
5 $A(z) = A(z) + P(z)B(z)$
6 $R(z) = R(z) + P(z)B(z)$
7 $Q(z) = Q(z) + P(z)$
8 **return** *quotient* $Q(z)$ *and remainder* $R(z)$

have to compute and store both of them. Second, the degree of the dividend polynomial could be as large as 10 which actually exceeds the maximum capacity of operand size, we have to adapt our polynomial divider to such case.

To solve the first problem that both quotient and remainder should be computed, the schoolbook approach — long division is applied. Let the dividend polynomial be $A(z)$, divisor polynomial be $B(z)$, the resulting quotient and remainder be $Q(z)$ and $R(z)$, **ldcoeff**$(\cdot)$ be the leading coefficient of a polynomial, **deg**$(\cdot)$ be the degree of a polynomial, then Algorithm 5 depicts the control flow for calculating the results of polynomial division such that $A(z) = Q(z) \cdot B(z) + R(z)$ where $deg(Q(z)) = deg(A(z)) - deg(B(z))$ and $deg(R(z)) < deg(B(z))$. Step 5 is the most critical operation to reduce the divisor $A(z)$ by calculating $A(z) = A(z) + \textbf{ldcoeff}(B(z))^{-1} \cdot \textbf{ldcoeff}(A(z)) \cdot z^{\textbf{deg}(A(z)) - \textbf{deg}(B(z))} \cdot B(z)$. This step can be further split into three substeps — α, β and γ shown in Eq. (3).

$$
\begin{aligned}
\alpha =& \textbf{ldcoeff}(B(z))^{-1} \\
\beta =& \alpha \cdot \textbf{ldcoeff}(A(z)) \\
\gamma =& \beta \cdot z^{\textbf{deg}(A(z)) - \textbf{deg}(B(z))} + A(z)
\end{aligned} \tag{3}
$$

The second problem is about the maximum length of the operand. This worst case comes when the dividend is the Goppa irreducible polynomial $g(z) = z^9 + g_8 z^8 + \ldots + g_0$ where 145-bits register is required to hold such value. However, it can be observed that by Eq. (3), the shift operation moves in the unit of 16 bits and this indicates when right shifting the 144-bit polynomial $B(z)$ for eliminating the leading coefficient 1 of $g(z)$, one needs a 160-bit register to hold the complete result. By default we use 144-bit registers to keep data and therefore 160-bit register is a waste of register utilities. We find through our analysis that 160-bit register is not necessary due to the nature of polynomial division. Let the 166-bit dividend polynomial be $A(z) = g(z) = z^9 + g_8 z^{t-1} + \ldots + g_0$, then according to Eq. (3), the divisor polynomial $B(z)$ multiplies by **ldcoeff**$(B(z))^{-1} \cdot$ **ldcoeff**$(A(z))$ and then shifts by $z^{9 - deg(B(z))}$. Note that after this multiplication and shift, the leading coefficient term of $B(z)$ changes to 1, denoted as $B'(z) = z^9 + \sum_{i=0}^{i=8} b'_i(z)$. This leads to the leading coefficient of the addition $g'(z) = g(z) + B'(z)$ equal zero and thus we do not necessarily need 160 bits register to calculate $g'(z)$ because in the end, the leading coefficient term of $g'(z)$ must be zero and thus we simply omit it.

The architecture of the proposed Goppa field polynomial divider can be found in Fig. 5. The divider initializes itself to state PRE and then moves to state DEGREE to calculate the polynomial degree of dividend $A(z)$ and divisor $B(z)$ using the module GOPF_DEG. Once the degrees are obtained, two things are simultaneously done in state LDCOEFF: One is to compute **ldcoeff**$(B(z))^{-1} \cdot$ **ldcoeff**$(A(z))$ in the partial product, the other is to prepare for $B(z) \cdot z^{deg(A(z)) - deg(B(z))}$ by right shifting $B(z)$ with $16 \times (deg(A(z)) - degB(z)))$ bits. Next the divider runs into state MAC to compute Eq. (3) using the MAC array in which we set the 16-bit multiplication operand to **ldcoeff**$(B(z))^{-1} \cdot$ **ldcoeff**$(A(z))$, the 144-bit multiplication operand to $B(z)z^{deg(A(z) - degB(z))}$ and the 144-bit addition operand to $A(z)$. Finally, the result from this MAC array updates the register $A(z)$ and the logic control runs back again into DEGREE to re-compute the degree of the newly generated $A(z)$. Afterwards, the state proceeds

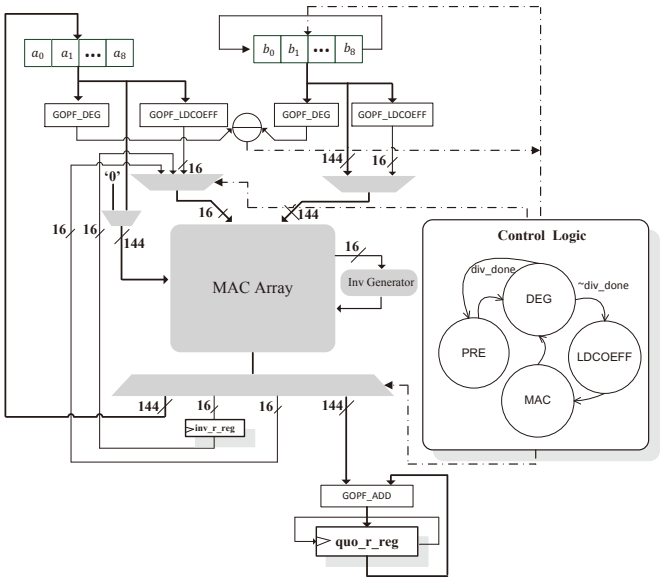

Fig. 5. Goppa field polynomial divider over $GF(2^{16})/g(z)$.

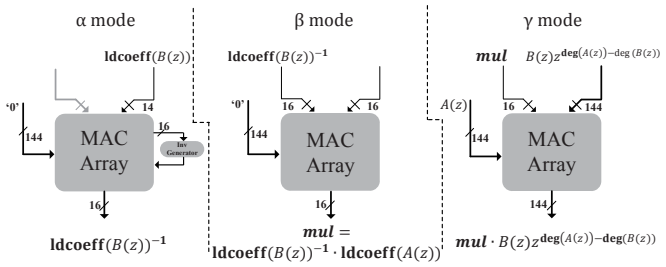

Fig. 6. Three different working modes of the MAC array inside the Goppa field polynomial divider — α mode computes $GF(2^{16})$ inverse, β mode computes $GF(2^{16})$ multiplication and γ mode computes scalar multiplication with addition. The data path highlighted in black indicates it is enabled, the data path in gray indicates it is disabled for connection.

repeatedly among DEGREE, LDCOEFF and MAC until the degree of $A(z)$ is smaller than that of $B(z)$.

It is worth mentioning that the design of the divider is more complex than that of the multiplier because the MAC array shared by the divider does more than scalar multiplication with addition: It also has to do $GF(2^{16})$ multiplication and inversion, as mentioned previously in Eq. (3). Nevertheless, we do not require additional computational resources but instead revise the MAC array to fulfil these operations. Fig. 6 depicts the three modes used to do $GF(2^{16})$ inversion, $GF(2^{16})$ multiplication and scalar multiplication with addition. In the first mode, the MAC array inputs $\mathbf{ldcoeff}(B(z)) \in GF(2^{16})$ and output its inverse. As discussed in Section II D.2, inverse can be done with one $GF(2^{16})$ multiplier and hence we utilize a single MAC module in the MAC array to serve as the multiplier and leave the other peripherals including the squaring block and the control logic part outside. Note that inverse is much more time-consuming than multiplication (See Table III) but we compute it for only once and store it in the register inv_r_reg. Next time when the divider recalls the inverse, the result can be directly retrieved from this register. This method is valid because the input of inverse — $B(z)$ remains constant during the process of division and thus the result $\mathbf{ldcoeff}(B(z))^{-1}$ also remains unchanged. In the second mode, we again exploit a single MAC to do multiplication like $\mathbf{ldcoeff}(B(z))^{-1} \cdot \mathbf{ldcoeff}(A(z))$. The third mode is the same as used in polynomial multiplication.

5) Goppa Field Polynomial Evaluator: When the error locator polynomial, also called sigma polynomial $\sigma(z)$, is obtained by the syndrome decoding i.e. Algorithm 1, the last remaining step is to retrieve the error vector from it. This process is actually to fist find all roots $\in L = (\alpha_0, \ldots, \alpha_{2^m-1})$ of polynomial $\sigma(z)$ and then map these roots to the error pattern, e.g. determine the set of error locations/indices $B = \{i|\sigma(\alpha_i) = 0\}$. Then the error pattern/vector $e = (e_0, \ldots, e_{n-1})$ is defined by $e_i = 1$ for $i \in B$ and $e_i = 0$ elsewhere. The proposed polynomial evaluator is such a module to compute the set B.

We first plan to implement the Berlekamp trace method [3], [16] known to be the fastest root finding algorithm for finite field polynomials. However, the Berlekamp's algorithm directly returns the roots but without their indices which is crucial to recover the error pattern. If we use the Berlekamp's algorithm, one has to solve discrete logarithm problem to retrieve these indices for which no efficient classical algorithms can do in polynomial time. Instead, the exhaust search could be more advantageous for obtaining those indices. More specifically, the exhaust search evaluates one by one all 2^{16} elements α_i of the support L to retrieve the desired 9 roots with their indices i such that $\sigma(\alpha_i) = 0$. In practice, we have analyzed two exhaust search algorithms: Horner's scheme [27] and Chien search [5]. It appears Chien search is more suitable for hardware implementations due to its considerably fast speed and much smaller memory overhead.

To explain how Chien search works, we first fix up the support $L = \{\alpha, \alpha^2, \ldots, \alpha^{2^m-1}, 0\}$ where α is a primitive element over $GF(2^m)$. Then the following relationship exists:

$$\begin{aligned}\sigma(\alpha^i) &= \sigma_0 + \sigma_1(\alpha^i) + \sigma_2(\alpha^i)^2 + \ldots + \sigma_t(\alpha^i)^t \\ &= \gamma_{0,i} + \gamma_{1,i} + \gamma_{2,i} + \ldots + \gamma_{t,i} \end{aligned} \tag{4}$$

$$\begin{aligned}\sigma(\alpha^{i+1}) &= \sigma_0 + \sigma_1(\alpha^{i+1}) + \sigma_2(\alpha^{i+1})^2 + \ldots + \sigma_t(\alpha^{i+1})^t \\ &= \gamma_{0,i} + \gamma_{1,i}\alpha + \gamma_{2,i}\alpha^2 + \ldots + \gamma_{t,i}\alpha^t \\ &= \gamma_{0,i+1} + \gamma_{1,i+1} + \gamma_{2,i+1} + \ldots + \gamma_{t,i+1}\end{aligned} \tag{5}$$

Putting it in another way, if each term $\gamma_{j,i}$ of $\sigma(\alpha^i)$ is known, then the terms $\gamma_{j,i+1}$ of its consecutive $\sigma(\alpha^{i+1})$ can be computed as $\gamma_{j,i+1} = \gamma_{j,i} \cdot \alpha^j$ where $0 \leq j \leq t$ and $0 \leq i \leq 2^m - 1$. Note that all t terms $\gamma_{j,i+1}$ can be computed simultaneously and therefore allows for a fast hardware implementation. On the other hand, Chien search is also very memory efficient because we solely store the initial $\sigma(\alpha^0) = \sum_0^t \sigma_i$ and the constant vector $(\alpha, \alpha^2, \ldots, \alpha^t)$, Eq. (5) iteratively calculates all the remaining series $\sigma(\alpha^i), i \neq 0$. Comparatively, Horner scheme requires to store the entire

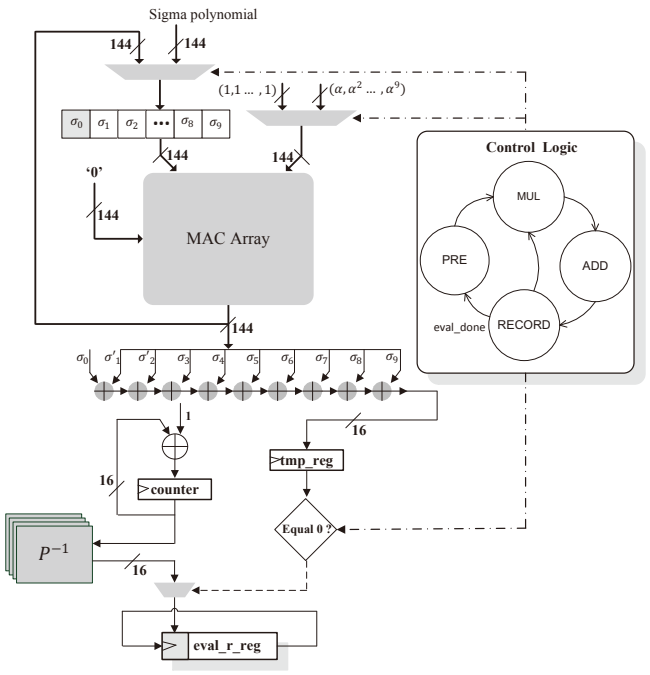

Fig. 7. Goppa field polynomial evaluator for $\sigma(z)$. The indices of the roots of sigma polynomial are recorded in eval_r_reg.

L set of 2^m $GF(2^m)$ elements, which is a costly memory requirement in our parameter setup with $m = 16$.

Fig. 7 depicts the Goppa field polynomial evaluator using Chein search. This evaluator evaluates one support element in three cycles. In the first cycle (state MUL), the MAC array outputs all 9 terms $\gamma_{j,i}$ of $\sigma(\alpha^i)$. In the second cycle (state ADD), these terms are summed up and stored into the 16-bit register — tmp_reg. Finally in state RECORD, the tmp_reg is evaluated to determine whether the input support element is the root of $\sigma(z)$ and if it is, the index of this element should be recorded to the 144-bit shift register — eval_r_reg. eval_r_reg is split into 9 sections and each is 16 bits because at most 9 roots exist and the indices of roots range from 0 to 65535 (16 bits). When the root is found, the index is then stored to the leftmost section and the other sections would shift right 16 bits simultaneously. In this way, we could compress the error pattern e by recording all the root indices into one 144-bit register. We also make a slight improvement to the previous implementation of Chein search [11] where we observe that the first term $\gamma_{0,i}$ of the series $(\sigma(\alpha), \sigma(\alpha^2), \ldots)$ remains constant during the iterations, e.g. $\sigma_0 = \gamma_{0,1} = \gamma_{0,2} = \ldots$. Therefore, it is unnecessary to implement another multiplier to calculate $\gamma_{0,i}, 0 \leq i \leq 2^{16}-1$ but instead to keep $\gamma_{0,i} = \sigma_0$ in the register. It is also worth stating that the MAC array we have designed is still applicable in the Chein search, but this time the working mode of the MAC array in the polynomial evaluator, different from the modes used in the previous modules, is purely the parallelized operation of 9 $GF(2^{16})$ multiplications. This operation inputs the 144-bit vector $(\sigma_1, \sigma_2, \ldots, \sigma_9)$ and $(\alpha, \alpha^2, \ldots, \alpha^9)$ and then outputs $(\sigma_1\alpha, \sigma_2\alpha^2, \ldots, \sigma_9\alpha^9)$ to update the vector $(\sigma_1, \sigma_2, \ldots, \sigma_9)$. Note that the positions of the error pattern are required to be permuted by multiplying P^{-1} (See step 4, Algorithm 3). In this context, we decide to merge this permutation with the root finding. As a side effect, the inverse permutation matrix P^{-1} with $2^{16} \times 2^{16}$ bits should be stored, which is too large to be accepted. To solve this, we implement P^{-1} as a look-up table in which the one-on-one permutation mapping is stored and thus the memory reduces to $2^{16} \times 16$ bits.

6) Others: Beside the instructions we have introduced above, there are still some others ALU instructions including *DEG*, *RSHIFT*, *UNSCRAMBLE* and *SPLIT*. *DEG* is used to calculate the degree of the polynomial and in our implementation this function is achieved by linear shift register. *RSHIFT* is the right shift of the polynomial by 16 bits. *UNSCRAMBLE* is for matrix-vector multiplication $S^{-1}c$, the second step in decryption (See Algorithm 3). This instruction executes by scanning each bit position of the vector c and summing up those columns of S^{-1} corresponding to the non-zero bit position of c. These four operations are simple and we now focus on the SPLIT operation.

In step 5, Algorithm 1, one needs to calculate the square root over $GF(2^m)[z]/g(z)$, e.g. $d(x) = \sqrt{T(z)+z} \bmod g(z)$. This is, in fact, an instance of a special-case square root and can be done easily if the modulus $g(z)$ and $T(z)+z$ are split into the following format:

$$g(z) = g_0^2(z) + zg_1^2(z), T(z) + z = T_0^2(z) + zT_1^2(z) \quad (6)$$

Then the square root $\sqrt{T(z)+z} \bmod g(z)$ is given by $T_0(z) + g_0(z)g_1^{-1}(z)T_1(z)$ [13], [26]. Hence to get this square root, we must implement the split operation (Eq. (6)) in the ALU. As Risse interprets in his paper [26], the split operation is accomplished first by splitting $g(z)$ or $T(z)+z$ into even odd parts (even and odd being a reference to the degree of each term) and then get the square root of the even part and odd part through a linear mapping. To clarify this point, we split $g(z) = z^9 + \sum_{i=0}^{8} g_i z^i$ for an instance. The even part and odd part are described as follows:

$$\begin{aligned} g_{even}(z) &= g_0 + g_2z^2 + g_4z^4 + g_6z^6 + g_8z^8 \\ &= (\sqrt{g_0} + \sqrt{g_2}z + \sqrt{g_4}z^2 + \sqrt{g_6}z^3 + \sqrt{g_8}z^4)^2 \\ &= g_0(z)^2 \end{aligned} \quad (7)$$

$$\begin{aligned} g_{odd}(z) &= g_1z + g_3z^3 + g_5z^5 + g_7z^7 + z^9 \\ &= z(\sqrt{g_1} + \sqrt{g_3}z + \sqrt{g_5}z^2 + \sqrt{g_7}z^3 + z^4)^2 \\ &= zg_1(z)^2 \end{aligned} \quad (8)$$

As we know, squaring any element over $GF(2^m)$ is a linear mapping, the inverse of this linear mapping yields the square root of elements over this particular $GF(2^m)$. In our design, we pre-compute such inverse matrix (16×16 binary entries) and implement it inside our GOPF_SPLIT module to compute $\sqrt{g_i}, 0 \leq i \leq 8$.

E. PRNG

Pseudo Random Number Generator (PRNG) is specifically designed to generate a sequence of random numbers for CFS signature scheme because in CFS scheme, the mechanism of

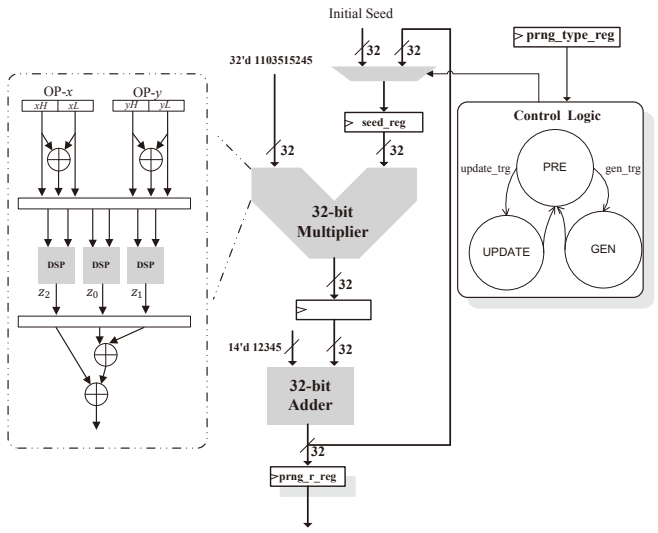

Fig. 8. PRNG module using linear congruent generator.

signature is the trials and errors for decoding and therefore unrepeated sequences of numbers are demanded. In this work, we apply LCG (Linear Congruent Generator) structure to produce pseudo-randomized numbers since it is easily implemented and fast on hardware.

LCG is defined by the recurrence relation:

$$X_{n+1} = (aX_n + c) \bmod m \tag{9}$$

where X is the sequence of pseudorandom values and a, c, m are the LCG parameters. In order to obtain a good LCG design which is capable of producing pseudo random numbers that can pass formal tests for randomness, one should pay extremely attention to the choice of the parameters a, c, m. In our work, we set $a = 1103515245, c = 12345, m = 2^{32}$ and output 30th bit $-$ 16th bit of X. This set of parameters is in common use and adopted by C99/C11 as ISO/IEC 9899 suggests.

The structure of LCG-based PRNG is illustrated in Fig. 8. PRNG works in two modes: On the one hand, the cryptoprocessor updates the seed of the random sequence by rewriting the seed_reg. On the other hand, LCG starts to output the random number stored in prng_r_reg by calculating Eq. (9) and then updates the seed_reg with the 30th bit $-$ 16th bit of X. It is worth mentioning that the 64-bit output from the 32-bit integer multiplier needs a modulus operation — $\mathrm{mod} 2^{32}$ and thus we just truncated the highest 32 bits and preserve the lowest 32 bit. The 32-bit multiplier itself is 5-stage pipelined and constructed using Karatsuba algorithm [36] to optimize the area. Karatsuba algorithm requires three 16-bit multipliers and we implement these multipliers by exploiting the existing DSP resources from Xilinx FPGA platform to boost performance.

IV. Experiments and Results

We present our Niederreiter cryptoprocessor implementation results in reconfigurable hardware — Xilinx Vertex-6 device. The most recent hardware implementation is from Heyse [11], we first compare and analysis our work with theirs. Then we focus on the CFS signature producing of the Niederreiter cryptosystem and compare our work with some other existing software implementation on PC given that no complete hardware implementation is available in the literature. Finally we give an overview of existing asymmetric cryptosystem implementations with quantum-computing resistance or non quantum-computing resistance on similar platforms and compare them to our results. All our results are obtained post place-and-route (PAR) from a Xilinx Virtex-6 XC6VLX240T FPGA using Xilinx ISE 13.2.

TABLE IV
Detailed FPGA Implementation results compared with [11] for roughly 80-bit security

Aspect (Virtex6-VLX240)	Niederreiter [11]	This work
Slices	3,887 (10%)	4,254 (11%)
LUTs	9,409 (6%)	10,718 (7%)
FFs	12,861 (4%)	8,624 (2%)
BRAMs	9 (2%)	299 (71%)
Frequency	250 MHz	250 MHz
Re-calculate $Syn(z)$	—	74 cycles
Undo Scrambling $c \cdot S^{-1}$	297 cycles	152 cycles
Compute $T = Syn(z)^{-1}$	4,310 cycles	1,035 cycles
Solve Equation	4,854 cycles	736 cycles
Test Decodibility	—	1,101 cycles
Find Roots	4,096 cycles	196,618 cycles
Others	283 cycles	—
Total Delay (Dec)	58.78 μs	798.57 μs
Total Delay (Sig $\lambda = 3$)	—	14.07 s

In Table IV, we provide with detailed comparing information between our work and Heyse's [11]. In [11], encryption and decryption are two sperate designs for the sake of less memory footprints and a more compact critical path. On the contrary, our design is a unified architecture for all cryptographic operations including encryption, decryption, signature and verification. Due to the significant large public key size resulting from our parameter selection, the memory consumption of our design is much larger than that of [11], i.e. 9 Mb vs 0.580 Mb. On the other hand, our design can perform signature scheme but [11] cannot. In terms of the encryption time, [11] outperforms ours with a factor of 13.58. However, this advantage relies mostly on the timing performance of the root finding: In [11], they only have to find 23 roots out of all 2048 candidates whereas we need to exhaust 55536 $GF(2^{16})$ elements before retrieving all 9 roots of $\sigma(z)$. In addition, it takes two clock cycles to evaluate a single element of the support L [11], our work requires three clocks because the permutation matrix P is much larger and an extra one clock is used for multiple staged addressing, as mention in Appendix A. The total delay of signature producing is obtained according to the average number of Goppa code decoding since the CFS signature scheme is actually a probabilistic algorithm and signature time varies for each run. We measure the running time of 1,000 signatures and use the average number of decoding $E = 1134855$ as the iteration number to compute the signature time.

The signature performance of our cryptoprocessor compared with other software implementation is shown in Table VI. Our cryptoprocessor can issue signatures of three security levels (54 bits, 69 bits, 75 bits) with 4.61 seconds, 9.37 seconds

TABLE V
PERFORMANCE COMPARISON OF OUR NIEDERREITER CRYPTOPROCESSOR IMPLEMENTATIONS WITH OTHER PUBLIC KEY SCHEMES FOR 80-BIT SECURITY

Scheme	Application	QC Resisted	Platform	Frequency (MHz)	Time (μs)	Slices/LUTs/FFs	BRAMs/DSPs
This work	Enc. Dec. **Sig.✓** Ver.	Yes	Virtex6-VLX240T	250	1.41 798.57 1.407×10^7 18.44	$4,254/10,718/8,624$	$299/3$
Niederreiter [11]	Enc. Dec.	Yes	Virtex6-LX240T	300 250	0.66 58.78	$315/926/875$ $3,887/9,409/12,861$	$17/0$ $9/0$
McEliece [32]	Enc. Dec	Yes	Virtex5-LX110T	163	500 1400	$14,537/-/-$	$75/0$
McEliece [12]	Enc. Dec.	Yes	Virtex6-VLX240T	351 191	13.66 85.79	$2,920/14,426/8,856$ $17,120/46,515/46,249$	$0/0$ $0/0$
NTRU [15]	Enc. Dec.	Yes	VirtexE-1600E	62.3	1.54 1.41	$14,352/5,160/27,292$	$0/1$
Ring-LWE [29]*	Enc. Dec.	Yes	Virtex6-LX75T	313 278	20.1 9.1	$-/1,349/860$	$2/1$
RSA-1024 [34]	Enc. Dec.	No	Virtex5-VLX30T	450	1520	$3,237/-/-$	$5/17$
ECC-163 [28]	Enc. Dec.	No	Virtex-4	45.5	12.1	$12,430/-/-$	$0/0$
ECC-163 [25]	Enc. Dec.	No	Virtex5-VLX85T	167	8.6	$3,446/10,176/-$	$0/0$

* [29] is assumed to achieve around 128-bit security whereas the other references listed here are assumed to achieve roughly 80-bit security.

and 14.07 seconds. Note that the parameter set $(2^{18}, 18, 9, 3)$ Landais et al. [17] have implemented is slightly more secure than $(2^{16}, 16, 9, 3)$ of ours ($2^{74.9}$ vs $2^{83.4}$) but this larger parameter set does not affect timing performance. This is because for issuing a signature, larger $m = 18$ indicates a four times larger public key size which roughly increases the total timing delay of root finding by four. However, the root finding is done for only once and takes very few proportion (0.00568%) of the total signature time. According to our estimation formula (Eq. (10), Appendix A), the run time is approximately 5.18% longer if we re-implement $(2^{18}, 18, 9, 3)$. As aforementioned though, we cannot directly implement their parameters at this moment because this extremely large public key size has already been beyond the maximum limit of block RAM resources on Viertex-6.

Table V overviews this work compared with published FPGA implementations of code-based (McEliece, Niederreiter), lattice-based (NTRU, Ring-LWE), and standard public key schemes (RSA, ECC) on the basis of around or above 80-bit security. Our design is the first prototypical architecture for a unified solution of encryption, decryption, signature and verification for the code-based cryptosystem. The current results show that encryption and verification can be done very fast. On the contrary, we have to significantly increase the size of the public key $\hat{H}$ and the support L in order to issue signatures, which in return results to a much longer decryption time when compared with [11]. On the other hand, the implementations of the lattice-based cryptosystem [15], [29] appear to offer a better time/area trade off. This is an important reason that lattice-based cryptography is now regarded as the most promising candidate for post-quantum cryptography. Nevertheless, as Bernstein suggests in his paper [2], NTRU will be patented until 2017, and Ring-LWE is less attractive than the McEliece cryptosystem for users concerned with the length and depth of cryptanalytic scrutiny. Moreover, [12] implements the McEliece using QC-MDPC code instead of the classical binary Goppa code to significantly reduce memory usage while maintaining the security level. Their results are even comparable to the Niederreiter implementation [11]. All these observations indicate a very promising trend of better area/time tradeoff for the code-based cryptography in the future.

TABLE VI
OUR FPGA PERFORMANCE COMPARED WITH SOFTWARE IMPLEMENTATION FOR CFS SIGNATURE SCHEME

Scheme	$(\mathbf{n}, \mathbf{m}, \mathbf{t}, \lambda)$	Platform	Frequency	Time
This work	$(2^{16}, 16, 9, 1)$	Virtex-6	250 MHz	4.61 s
	$(2^{16}, 16, 9, 2)$			9.37 s
	$(2^{16}, 16, 9, 3)$			14.07 s
Landais [17]	$(2^{18}, 18, 9, 3)$	Xeon-W3670	3.2 GHz	14.70 s
	$(2^{18}, 18, 9, 4)$			19.61 s
Courtois [6]	$(2^{16}, 16, 9, 1)$	N/A	1 GHz	10 s

V. CONCLUSION AND FUTURE WORK

In this work we presented a new implementation for the Niederreiter cryptosystem on Xilinx Virtex-6 FPGAs. Our implementation is primarily designed for a generic architecture for all basic cryptographic operations including encrypting/decrypting data and signing/verifying signatures. Current Niederreiter implementations focus on encryption/decryption but leave the signature issuing aside since signing a signature

is complicated and difficult to be integrated. We analyzed the common things between decryption and signature, proposed a new secure parameter selection method, and designed a unified processor architecture for it. Finally we evaluated the performance of our design compared with other cryptosystem implementations for the corresponding platforms. The results showed that it is indeed possible to completely realize the Niederreiter signature scheme under which signature issuing and data encryption co-exist well without redesign of circuits on reconfigurable hardware with acceptable time/area trade-offs.

Despite its acceptable features, our prototypical crypto-processor does not have the function to convert any binary stream of plaintext into the form of constant weight words at the very first step of data encryption and convert it back at the end of data decryption, though it requires very few clock cycles to complete. Constant weight means that the Niederreiter cryptosystem requires the number of '1' in the plaintext must be smaller than or equal to t. [11] proposed to adapt the constant weight coding/decoding method from [30] for embedded system applications. However, we find such method is not applicable for large n because their design maintains a $n \times t$ lookup table to encode input messages into constant weight words and this table is too large to be accepted on Xilinx FPGAs when using large parameter set like ours. We plan to optimize this constant weight encoder by reducing memory overhead in future.

REFERENCES

[1] D. J. Bernstein. Introduction to post-quantum cryptography. In *Post-quantum cryptography*, pages 1–14. Springer, 2009.

[2] D. J. Bernstein, T. Lange, and C. Peters. Smaller decoding exponents: ball-collision decoding. In *Advances in Cryptology–CRYPTO 2011*, pages 743–760. Springer, 2011.

[3] B. Biswas and V. Herbert. Efficient root finding of polynomials over fields of characteristic 2. 2009.

[4] D. D. Chen, N. Mentens, F. Vercauteren, S. S. Roy, R. C. Cheung, D. Pao, and I. Verbauwhede. High-speed polynomial multiplication architecture for ring-lwe and she cryptosystems. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 62(1):157–166, 2015.

[5] R. T. Chien. Cyclic decoding procedures for bose-chaudhuri-hocquenghem codes. *IEEE transactions on information theory*, 10(4):357–363, 1964.

[6] N. T. Courtois, M. Finiasz, and N. Sendrier. How to achieve a mceliece-based digital signature scheme. In *Advances in Cryptology-ASIACRYPT 2001*, pages 157–174. Springer, 2001.

[7] J.-P. Deschamps. *Hardware implementation of finite-field arithmetic*. McGraw-Hill, Inc., 2009.

[8] M. Finiasz. Parallel-cfs. In *Selected areas in cryptography*, pages 159–170. Springer, 2011.

[9] M. Finiasz and N. Sendrier. Security bounds for the design of code-based cryptosystems. In *Advances in Cryptology–ASIACRYPT 2009*, pages 88–105. Springer, 2009.

[10] S. Heyse. Low-reiter: Niederreiter encryption scheme for embedded microcontrollers. In *Post-Quantum Cryptography*, pages 165–181. Springer, 2010.

[11] 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*, pages 340–355. Springer, 2012.

[12] S. Heyse, I. Von Maurich, and T. Güneysu. Smaller keys for code-based cryptography: Qc-mdpc mceliece implementations on embedded devices. In *Cryptographic Hardware and Embedded Systems-CHES 2013*, pages 273–292. Springer, 2013.

[13] K. Huber. Note on decoding binary goppa codes. *Electronics Letters*, 32(2):102–103, 1996.

[14] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses GF(2m) using normal bases. *Information and computation*, 78(3):171–177, 1988.

[15] A. A. Kamal and A. M. Youssef. An fpga implementation of the n-truencrypt cryptosystem. In *Microelectronics (ICM), 2009 International Conference on*, pages 209–212. IEEE, 2009.

[16] J. Kerl. The berlekamp algorithm. 2009.

[17] G. Landais and N. Sendrier. Cfs software implementation. *IACR Cryptology ePrint Archive*, 2012:132, 2012.

[18] E. D. Mastrovito. Vlsi designs for multiplication over finite fields gf (2m). In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 297–309. Springer, 1989.

[19] E. D. Mastrovito. *VLSI architectures for computations in Galois fields*. Linkoping univ. Dep. of electrical engineering Link@: oping, 1991.

[20] A. May, A. Meurer, and E. Thomae. Decoding random linear codes in\ tilde {\ mathcal {O}}(2^{0.054 n}). In *Advances in Cryptology–ASIACRYPT 2011*, pages 107–124. Springer, 2011.

[21] R. J. McEliece. A public-key cryptosystem based on algebraic coding theory. *DSN progress report*, 42(44):114–116, 1978.

[22] H. Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *PROBLEMS OF CONTROL AND INFORMATION THEORY-PROBLEMY UPRAVLENIYA I TEORII INFORMATSII*, 15(2):159–166, 1986.

[23] N. J. Patterson. The algebraic decoding of goppa codes. *Information Theory, IEEE Transactions on*, 21(2):203–207, 1975.

[24] T. Pöppelmann and T. Güneysu. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In *Progress in Cryptology–LATINCRYPT 2012*, pages 139–158. Springer, 2012.

[25] C. Rebeiro, S. S. Roy, and D. Mukhopadhyay. Pushing the limits of high-speed gf (2 m) elliptic curve scalar multiplication on fpgas. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 494–511. Springer, 2012.

[26] T. Risse. How sage helps to implement goppa codes and mceliece pkcss. *Trans. Inform. Theory. Vol. IT*, 15(1):122–127, 1969.

[27] R. L. Rivest and C. E. Leiserson. *Introduction to algorithms*. McGraw-Hill, Inc., 1990.

[28] S. S. Roy, C. Rebeiro, and D. Mukhopadhyay. A parallel architecture for koblitz curve scalar multiplications on fpga platforms. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 553–559. IEEE, 2012.

[29] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede. Compact ring-lwe cryptoprocessor. In *Cryptographic Hardware and Embedded Systems–CHES 2014*, pages 371–391. Springer, 2014.

[30] N. Sendrier. Encoding information into constant weight words. In *Information Theory, 2005. ISIT 2005. Proceedings. International Symposium on*, pages 435–438. IEEE, 2005.

[31] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM journal on computing*, 26(5):1484–1509, 1997.

[32] A. Shoufan, T. Wink, H. G. Molter, S. A. Huss, and E. Kohnert. A novel cryptoprocessor architecture for the mceliece public-key cryptosystem. *Computers, IEEE Transactions on*, 59(11):1533–1546, 2010.

[33] J. Stern. A method for finding codewords of small weight. In *Coding theory and applications*, pages 106–113. Springer, 1989.

[34] 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*, 94(1):211–222, 2011.

[35] D. Wagner. A generalized birthday problem. In *Advances in cryptologyCRYPTO 2002*, pages 288–304. Springer, 2002.

[36] A. Weimerskirch and C. Paar. Generalizations of the karatsuba algorithm for efficient implementations. *IACR Cryptology ePrint Archive*, 2006:224, 2006.

APPENDIX

A. Proposed Run Time Evaluation

In this section we explain the proposed run time formula for the Niederreiter scheme evaluation. The run time of the encryption and verification is negligible and thus we focus on the average run time of decryption and signature. Decryption is principally about the syndrome decoding of Goppa code (Algorithm 1), which can be decomposed into three main parts:

1) Obtain the inverse of the syndrome $s(z)$.
2) Solve the equation $d(z)b(z) \equiv a(z) \bmod g(z)$ to get $\sigma(z)$.
3) Find all roots of $\sigma(z)$. For signature producing, one has to additionally apply the divisibility test.

Inverse of $s(z)$ is done by extended Euclidean algorithm requiring about t polynomial multiplications and t polynomial divisions. The equation $d(z)b(z) \equiv a(z) \bmod g(z)$ can be solved by a half extended Euclidean algorithm with about t polynomial multiplications and $\frac{t}{2}$ polynomial divisions. The divisibility test consists of m polynomial divisions and the root search takes $n \cdot t$ $GF(2^m)$ multiplications. Assume that one polynomial multiplication takes t^2 $GF(2^m)$ multiplications and one polynomial division takes $\frac{t^2}{2}$ $GF(2^m)$ multiplications, then the following formula describes the average run time of the Niederreiter cryptosystem (counted in the number of $GF(2^m)$ multiplications):

$$RunTime \approx \theta(\frac{11}{4}t^3+nt)+(1-\theta)(\frac{11}{4}t^3+mt^2)t!+nt \quad (10)$$

Where θ is the probability that data decryption takes place and we set it to be 0.5 in Table I.

B. Instruction Format

All instructions of the cryptoprocessor are listed in Table VII. They are categorized into four groups: data transfer, arithmetic and logic, random number generation and transfer of control. The most fundamental type of the proposed machine instruction is the data transfer instruction. In our cases, we specify the following rules: First, the location of the source and destination operands must be either registers to memory or memory to registers. Second, the length of data to be transferred is 144 bits because the algebraical decoding of Goppa code, which is the most critical process of the Niederreiter cryptosystem, handles 144-bit polynomials. Third, the addressing modes for each operand include immediate, direct and register. Immediate addressing (*MOV #imm R[x]*) is the simplest form of addressing, in which the operand value imm is present in the instruction and then moves into register R[x]. With direct addressing (*MOV R[x] @addr, MOV R[x] R[y], MOV R[x] IDX[y]*, the processor can transfer data directly between memory and registers, or registers and registers. Indirect addressing (*MOV @IDX[y] R[x], MOV R[x] @IDX[y]*) is necessary because in CFS signature scheme, the pseudo random number generator (PRNG) randomly outputs the memory address and one cannot obtain it during the period of instruction decoding and therefore, the processor must first store the address in registers and then address the memory according to the value stored in the registers.

The cryptoprocessor provides the basic arithmetic operations of polynomial addition, multiplication, division and $GF(2^{16})$ inversion. Moreover, extra useful instructions including *DEG, SPLIT, RSHIFT, UNSCRAMBLE, and EVAL* are particularly implemented for decoding Goppa Code. Details of these instructions are discussed in Section III D.

Two instructions related to random number generation are *PRNG #imm R[x]* and *PRNG R[x]*. The first one is to update the seed of the PRNG with imm and the second is to start the PRNG and output the random number to register R[x].

For all of the operation types discussed so far, the next instruction to be performed is the one that immediately follows the current instruction. However, in practical use of the Niederreiter cryptosystem, it is essential to execute instructions more than once and most likely many thousands of times to significantly reduce memory footprint. In addition, the Niederreiter cryptosystem involves some decision making. For instance, in syndrome decoding (see step 1 and step 6, Algorithm 1), one needs a while-loop structure to find the inverse of a polynomial and to solve the equation $d(x)b(x) \equiv a(x) \bmod g(x)$. To fulfil these two requirements mentioned above, we design two branch instructions, also called jump instructions which have as one of its operands the address of the next instruction to be executed. One is the conditional branch instruction and symbolized as *JRE IDX[x] @label*. The other is the unconditional branch instruction and symbolized as *JMP @label*.

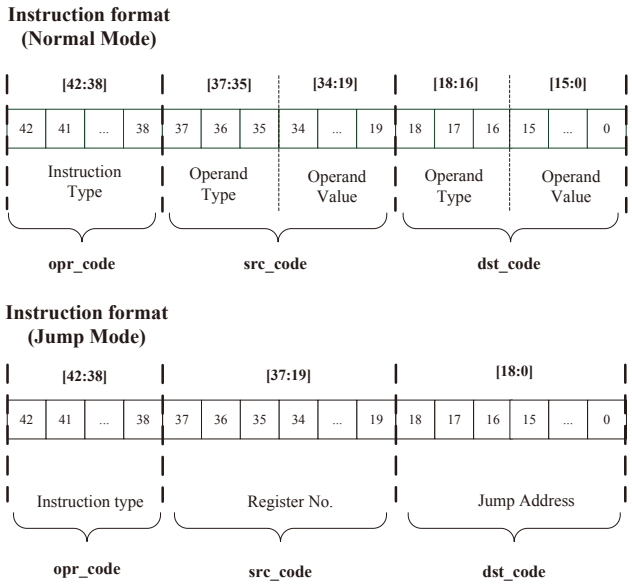

Fig. 9. Two instruction modes used in our design.

To translate all existing operations mentioned in Table VII into binaries for our processor to proceed, the length of the proposed instructions sets to be 43 bits and has three segments: opr_code (5 bits), src_code (19 bits) and dst_code (19 bits). opr_code indicates the operand type executed by the processor. src_code and dst_code store the first operand and the second operand respectively. After the execution of the instruction, the result is written back to the second operand. In other words, dst_code plays a dual role — On the one hand, it stores the second operand prepared for processing, on the other hand, it also stores the result obtained after processing. In Fig. 9, we list the instruction formats with two modes used in our design. One is called normal mode for sequential execution of instructions, including *MOV, ADD, INV, MUL, DIV, SPLIT, DEG, RSHIFT, EVAL, PRNG, IDX* and the other is called jump mode including *JMP, JRE*. The first 5 bits in the instruction word distinguish different operation types (See Table VIII).

TABLE VII
EXTENDABLE INSTRUCTION SET OF THE ASIP

Type	Instruction	Illustration	Latency (cycles)	Example
Data Transfer	MOV @IDX[y] R[x]	Move data from external memory (addr=IDX[y]) to register R[x]	8	'MOV @IDX0 R1' means to move data from external memory at addr=IDX0 to register R1
	MOV R[x] @addr	Move data from register R[x] to external memory	4	'MOV R[x] @4' means to move data at register R[x] to external memory at addr=4
	MOV #imm R[x]	Move an immediate data to register	4	'MOV #11111111 R2' means to move 11111111 to register R2
	MOV R[x] R[y]	Move data from register R[x] to register R[y]	4	'MOV R2 R0' means to move data at reg R2 to reg R0
	MOV R[x] @IDX[y]	Move data from R[x] to external memory (addr=IDX[y])	5	'MOV R0 @IDX0' means to move data R0 to external memory at addr=IDX0
	MOV R[x] IDX[y]	Move R[x] into IDX[y]	4	'MOV R13 IDX1' means to move data at R13 to IDX1
Arithmetic and Logic	ADD R[x] R[y]	$R[y]' = R[x] + R[y]$	7	'ADD R0 R2' means to add R0 and R2 and stor ethe result into R2
	INV R[x] R[y]	$R[y]' = R[x]^{-1}$	34	'INV R0 R0' means to calcuate the inverse of R0 and store the result into R0
	MUL R[x] R[y]	$R[y]' = R[x] \cdot R[y] \bmod g(z)$	≤ 89	'MUL R2 R2' means to multiply R2 by R2 and store the result into R2
	DIV R[x] R[y]	$R[x]' = R[x]/R[y]$, $R[y]' = R[x] \bmod R[y]$	≤ 69	'DIV R2 R3' means to divide R2 by R3 and then store quotient to R2 and remainder to R3
	SPLIT R[x] R[y]	Split $R[x]$ such that $R[x] = R[x]'^2 + zR[y]'^2$	26	'SPLIT R2 R3' means to calculate R2 and R3 such that $R2 = R2^2 + z * R3^2$
	DEG R[x] R[y]	$R[y] = degree(R[x])$	≤ 17	'DEG R2 R3' means to calculate deg(R2) and store it in R3
	RSHIFT R[x] R[y]	right shift R[x] and store it in (R[x],R[y])	7	'RSHIFT R2 R3' means to right shift R2 and store the MSB part in R2 and the remaining part in R3
	UNSCRAMBLE R[x] R[y]	$R[y] = S^{-1}R[x]$	152	'UNSCRAMBLE R2 R3' means to unscramble the ciphertext R[x] by multiplying the matrix S^{-1} and store the result in R3
	EVAL R[x] R[y]	$R[y]' = R[x](R[y])$	68	'EVAL R2 R3' means to evaluate R2 by substituting the unknowns with R3
Random Number Generation	PRNG #imm R[x]	Update PRNG using imm as its seed	6	'PRNG 32 R2' means to update PRNG with seed=32 and update R2=32
	PRNG R[x]	Start PRNG and transfer the random number into R[x]	6	'PRNG R20' means to generate random number and move it into R20
Transfer of Control	JMP @label	Unconditional jump to label	6	'JMP@4' means to jump to program at line 4 unconditionally
	JRE IDX[x] @label	If IDX[x] is bigger than R7, then jump to label line	9	'JRE IDX2 @1' means to jump to program at line 1 if IDX2 $>$ R31
	IDX[x]++	$IDX[x]' = IDX[x] + 1$	4	'IDX1++' means to update IDX1 by adding '1'
	IDX[x]−−	$IDX[x]' = IDX[x] - 1$	4	'IDX2−−' means to update IDX2 by subtracting '1'
	HALT	Halt the ASIP	1	—

In normal mode, for each instruction, two operands are required (For single operand instructions like *INV, DEG*, only dst_code is used.) — They can be

1) both registers
2) one register and one memory block
3) one immediate data and one register

3-bits of operand type in our instruction format is left for distinguishing different operand types (Table VIII).

Jump mode works for JMP and JNZ only. For JMP, the src_code is left blank but the dst_code records the destination address which your program should jump to. For JRE, the src_code is used to indicate which SPRF register compares with GPRF register R7 and the dst_code records the destination address.

C. Register Files

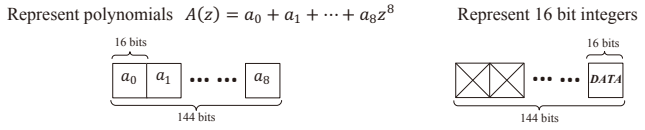

Fig. 10. Two types of data represented in registers and memory. When interpreting polynomials, the leftmost part is the polynomial coefficient with the lowest degree and the rightmost part holds the highest. On the other hand, when representing integers, the rightmost part holds the valid value while the remaining part becomes invalid.

In the Niederreiter cryptosystem, the computation is mainly about the polynomials arithmetic over $GF(2^m)$ and thus GPRF (Fig. 11) is used to store these polynomials. According to our parameter selection, the polynomials are over $GF(2^{16})$

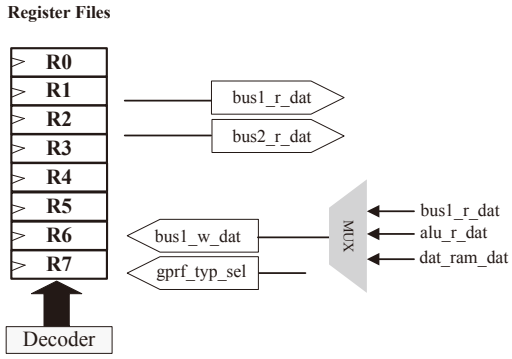

Fig. 11. GPRF module. It consists of 8 × 144-bit registers with two read ports (bus1_r_dat and bus2_r_dat) and one write port (gprf_w_dat). Two read ports — bus1_r_dat and bus2_r_dat are implemented because the majority proportion of instruction execution of our program relates to ALU arithmetic operations and for most of the time, ALU is fed up with two registers as its inputs and thus it is more timing efficient to have two independent read ports to transfer data to ALU.

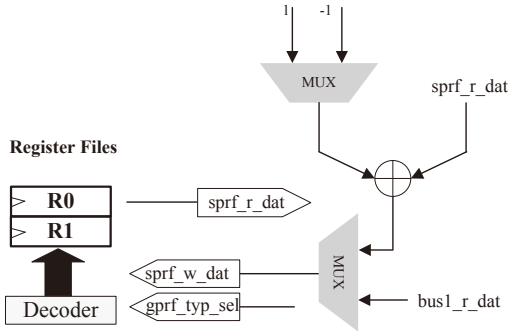

Fig. 12. SPRF module. It consists of 2 registers with one read port (sprf_r_dat) and one write port (sprf_w_dat) for reading and writing. Additionally, GPRF can increase or decrease its values by adding 1 or subtracting 1. This operation is essential because in Niederreiter cryptosystem, decodability test and root finding are programmed in for-loop and the loop index is exactly kept and updated in SPRF.

TABLE VIII
BINARY INTERPRETATION OF THE ASSEMBLY CODES PROPOSED

opr_code (5 bits)	Binaries
MOV	00001
ADD	00010
SUB	00011
MUL	00100
DIV	00101
PRNG	00110
SPLIT	00110
DEG	00110
RSHIFT	00110
EVAL	00110
IDX	00111
INV	01000
HALT	00000
JMP	10000
JRE	10001

operand type of src_code/dst_code (3 bits)	Binaries
register (gprf/sprf)	000
memory	001
immediate data	010
gprf mod register	100
indirect sprf register	101

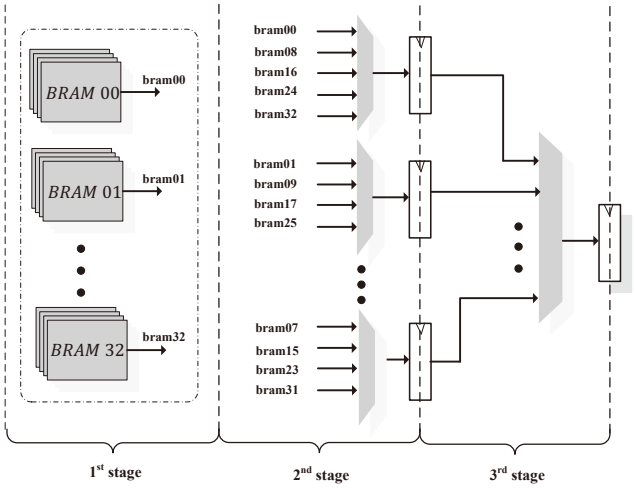

Fig. 13. Internal memory re-arrangement using multiple staged address decoding, for improving the timing performance of the proposed processor.

and have degree of 9, which indicates that the data width of the GPRF should be $16 \times 9 = 144$ bits. Additionally, integer numbers such as memory addresses, polynomial degrees and random numbers are also required and must be presented in the GPRFs. In order to solve this, we unify the way in which both types of data are represented shown in Fig. 10. On the other hand, SPRF (Fig. 11) is designed to be a much smaller register file compared with GPRF (2 registers vs 8 registers) but capable of handling for-loop structure in our program. In our instruction set, GPRF refers exactly to $R[x], 0 \leq x \leq 7$ and SPRF to $IDX[x], 0 \leq x \leq 1$.

D. Data RAM Management

We propose in this section a new way of organizing the data memory component to preserver high frequency of the circuits. Unlike other popular public key cryptosystems, the Niederreiter cryptosystem requires large memory to store the public key and some other useful data. Traditional methods to implement these memory on reconfigurable hardware, especially when using our parameter selection of the Goppa code, cannot achieve a considerable circuit frequency for real world applications. In our design three pieces of BRAMs are used to implemented the external main memory. The first is the main one, storing the public key — $\hat{H}_{(2^{16} \times 144)}$/secret key — Goppa polynomial $g(z)$, and the input plaintext/ciphertext. The other two are for storing the inverse permutation matrix $P^{-1}_{(2^{16} \times 2^{16})}$ and for the unscrambling matrix $S^{-1}_{(144 \times 144)}$ respectively. The number of address entries of the first two RAMs is so large that a direct instantiation of this part as a single BRAM cannot maintain the high frequency of our design. To improve its timing performance, we decide to use 33 pieces of 288 Kb BRAMs together to make up the main memory with an additional address decoder. We also use a similar architecture for the matrix P^{-1}.

Fig. 13 depicts the main memory management in which three staged address decoding is applied. In the first stage, the 33 BRAMs, each with 2048 addresses, concurrently output 33 possible ways of data. These data are then filtered to 8 ways of data through multiplexer and decoders. Eventually, the correct data with the exact address becomes valid in the third stage after the final 8-to-1 decoder. The experiments have shown that with this measure, our processor is able to function at the frequency of 250 MHz on Virtex-6 FPGAs.

E. Example Program

In this section, we introduce some important subroutines in our instruction memory to demonstrate how these instructions are organized to achieve data decryption or signature. These subroutines include XGCD, Modified_XGCD and DECODE_TEST shown in Fig. 14, 15, 16. The first two are the key operations in Goppa code decoding. The last is for signature scheme only.

```
MOV #0 R7
XGCD: MOV R1  R5
DIV R0  R1
MOV R0 R6
MOV R5 R0
MOV R6 R4
MUL R3 R4
ADD R2 R4
MOV R3 R2
MOV R4 R3
DEG R1 R6
MOV R6 IDX0
JRE IDX0    @XGCD
INV R1 R1
MUL R1  R3//T(z) = s(z)^(-1)
```

Fig. 14. The subroutine XGCD is for calculating $T(z) = s(z)^{-1} \bmod g(z)$ using extended Euclidean algorithm.

XGCD is used to perform the standard Euclidean algorithm [7] to obtain the inverse of the syndrome polynomial. Note that XGCD loop does not terminate at $R1 = 1$, instead it terminates at $deg(R1) = 0$. In other words, we have to calculate the inverse of R1 before eventually retrieving the correct $T(z)$. This phenomenon is further described and discussed in detail by [32].

The subroutine MODIFIED_GCD is, first of all, to solve the equation $d(z)b(z) \equiv a(z) \bmod g(z)$ in step 4, Algorithm 1 and then to return the sigma polynomial $\sigma(z) = a^2(z) + zb^2(z)$. The extended Euclidean algorithm can be applied to solve this equation with minor modifications: MODIFIED_GCD loop terminates when $deg(R1) \leq 4$ for the first time whereas XGCD does when $deg(R1) = 0$. The last two steps of normalization in this subroutine is to correct $\sigma(z)$ to be a monic polynomial for adjusting to the 144-bit data width of our processor.

The subroutine DECODE_TEST is used to determine the validity of $\sigma(z)$ by checking whether or not $z^{2^m} = z \bmod \sigma(z)$, mentioned in Section II C. This modular exponentiation is realized by m times of repeated squarings —*MUL R1 R1* with initial value $R1 = z$.

```
MOV #4 R7
MODIFIED_XGCD:  MOV R1  R5
DIV R0  R1
MOV R0 R6 // q in R6
MOV R5 R0 //update R0,R1
MOV R6 R4  // q in R4
MUL R3 R4
ADD R2 R4
MOV R3 R2
MOV R4 R3 //update R2,R3
DEG R1   R6
MOV R6  IDX0
JRE IDX0   @MODIFIED_XGCD
MUL R1 R1 //alpha^2
MUL R3 R3 //beta^2
RSHIFT R3 R0 //{R3,R0}=X*beta^2
ADD R3 R1 //{R1,R0}=sigma
INV R0  R0
MUL R1 R0 //normalize sigma
```

Fig. 15. The subroutine MODIFIED_GCD is to solve the equation in step 4, Algorithm 1 and then to return the sigma polynomial $\sigma(z)$.

```
DECODE_TEST: MUL R1 R1 //squaring
IDX0--
JRE IDX0   @DECODE_TEST
MOV R6  IDX0
MOV @IDX0  R2 //load 'x' to R1
ADD R2 R1 // X^2m + X
MOV R1   IDX0
JRE IDX0    @DECODEFAILURE
```

Fig. 16. The subroutine DECODE_TEST is used to determine the validity of $\sigma(z)$ by checking whether or not $z^{2^m} = z \bmod \sigma(z)$

In practice, we use 14 instructions for data encryption, 105 for data decryption, 119 for signature release and 16 for signature verification. A single piece of 5 Kb instruction SRAM is good enough for all these applications with our parameter selections.