RVCE-FAL: A RISC-V Scalar-Vector Custom Extension for Faster FALCON Digital Signature

Xinglong Yu, Yi Sun, Yifan Zhao, Honglin Kuang and Jun Han
State Key Laboratory of Integrated Chips and Systems, Fudan University, Shanghai, China
Email: junhan@fudan.edu.cn

***Abstract*—The National Institute of Standards and Technology (NIST) has selected FALCON as one of the standardized digital signature algorithms against quantum attacks in 2022. Compared with the other post-quantum cryptography (PQC) schemes, lattice-based FALCON is more appropriate for future Internet of Things (IoT) applications due to the fastest signature verification process and the lowest transmission overhead. In this paper, we propose a custom extension based on the RISC-V scalar-vector framework for efficient implementation of FALCON. To our best knowledge, this work is the first hardware-software co-design for complete FALCON signature generation and verification routines. Besides, we design the first FALCON Gaussian sampling hardware and a RISC-V vector extension (RVV) based domain-specific core. The proposed architecture accelerates kernel operations in FALCON, such as discrete Gaussian sampling, number theoretic transform (NTT), inverse NTT, and polynomial operations. Compared with the reference implementation, results on the gem5-RTL simulation platform present a speedup for signature generation and verification of up to 18× and 6.9×.**

***Index Terms*—post-quantum cryptography, FALCON, RISC-V custom extension, discrete Gaussian sampling, vector architecture**

I. Introduction

With the development of quantum computers, most existing public-key cryptography (PKC) infrastructures have been proved unsafe because of efficient quantum algorithms like Shor's algorithm [1]. Specifically, the security of traditional PKC schemes like RSA [2] and ElGamal [3] relies on the practical difficulty of integer factorization and discrete logarithms, which can be broken by a powerful quantum computer.

Under this circumstance, the National Institute of Standards and Technology (NIST) has been standardizing quantum-secure PKC algorithms since 2016. In 2022, NIST announced four post-quantum cryptography (PQC) finalists, CRYSTALS-Kyber, CRYSTALS-Dilithium, FALCON, and SPHINCS+ [4], as next-generation cryptosystems. In 2023, NIST released draft standards for the other three schemes except for FALCON, and it will be available for FALCON in 2024. Considering applications of FALCON are close to practice, efficient implementation in software and hardware is vitally significant for electronic and embedded systems.

Compared to CRYSTALS-Dilithium and SPHINCS+, FALCON [5] digital signature scheme exceeds in verification speed and communication overhead due to the smaller combined size of public key and signature, making it more suitable for future Internet of Things (IoT) applications [6]. The more immediate response will empower IoT devices. For faster applications of FALCON in terminal devices, it's more necessary to accelerate the frequent digital signature generation and verification rather than the unusual public/private key pair generation.

There have been some hardware implementations for faster FALCON. The researches in [6] [7] only focus on accelerating FALCON signature verification with pure hardware. However, flexibility is equally important because the draft standard has yet to be published. Furthermore, the hardware and software (HW/SW) co-design in [8] also concentrates on the FALCON verifying based on RISC-V. Nevertheless, they lack signature generation optimization. Besides, the design in [9] only focuses on the discrete Gaussian sampling in signature generation and needs to provide a comprehensive method for faster signing.

There are two challenges for accelerating FALCON over NTRU (number theory research unit) lattices with a fast Fourier trapdoor sampling. One is that the execution time of polynomial operations over rings accounts for roughly 61% of signature verification. The computation speed of polynomial arithmetic can be improved by exploring the data-level parallelism (DLP). The other is that the discrete Gaussian sampler needed in the fast Fourier sampling requests support for variable floating-point mean and standard deviation with high-precision demand. The discrete Gaussian sampler also requires efficiency because of the high computation overhead, representing about 35% in FALCON signing with floating-point function units.

In this paper, we propose RVCE-FAL, the first comprehensive FALCON acceleration solution founded on RISC-V scalar-vector custom instructions for entire signature generation and verification routines. The flexible and scalable RISC-V vector extension (RVV), currently after the public review in a stable version [10], inherits the advantages of RISC-V to allow fine-grained vectorized programming. We exploit DLP in a single-instruction-multiple-data (SIMD) way to accelerate polynomial operations using enhanced RVV. Moreover, we design efficient custom hardware, such as the first FALCON Gaussian hardware sampler, to break the computational bottleneck. The key contributions of this paper are listed as follows:

- We propose custom instructions to support the vectorized algorithms and improve the computational efficiency of FALCON. We thoroughly explore the DLP of the kernel operations, such as number theoretic transform (NTT), inverse NTT, fast Fourier transform (FFT), inverse FFT, and polynomial functions in RISC-V vector architecture.

This work was supported by the National Natural Science Foundation of China under Grant 61934002 and 62234008.

979-8-3503-4859-0/DATE24/© 2024 EDAA

- We design custom hardware as function units for RVCE-FAL. The proposed hardware enables custom instructions to accelerate modular arithmetic, SHA-3 functions, and discrete Gaussian sampling through HW/SW co-design. The proposed discrete Gaussian sampler is the first full hardware implementation so far. Finally, we also evaluate the design on the Xilinx UltraScale+ ZCU104 platform.
- We integrate RVCE-FAL into an RVV-enabled gem5-RTL simulation platform. The time performance of FALCON kernel operations and signature processing has been obtained. Compared with the reference C implementation, RVCE-FAL achieves up to 18× and 6.9× speedup for signature generation and verification routines respectively.

II. Preliminaries

A. Symbol Notation

For an integer q, the quotient ring $\mathbb{Z}/q\mathbb{Z}$ is denoted by $\mathbb{Z}_q$. In FALCON, $\mathbb{Z}_q$ is a finite field because the modulus $q = 12289$ is prime. Bold uppercase letters like $\mathbf{B}$ denote matrices and bold lowercase letters like $\mathbf{v}$ denote vectors. Vectors also include polynomials denoted by italic letters like f and G. The notation $\|s\|$ represents the 2-norm of a vector s. We define $\mathcal{Q}$ and $\mathcal{R}_q$ to present the rings $\mathbb{Q}[x]/(\phi)$ and $\mathbb{Z}_q[x]/(\phi)$. FALCON uses the polynomial modulus $\phi = x^n + 1$. There are two variants of FALCON, FALCON-512 and FALCON-1024, with degree $n = 512$ and 1024, for NIST security level I and V.

B. FFT and NTT Algorithms

Fast Fourier transform (FFT) and number theoretic transform (NTT) are highly applied for efficient calculations. These two algorithms share the same form of the evaluating expression, defined in equation (1). However, the difference is the twiddle factor ω_n^{ik}, the n-th primitive root of unity in the complex field for FFT or the finite field for NTT.

$$\hat{a}_k = \sum_{i=0}^{n-1} a_i \cdot \omega_n^{ik} = f(\omega_n^k), \text{ where } f(x) = \sum_{i=0}^{n-1} a_i \cdot x^i \quad (1)$$

To reduce the complexity of the polynomial multiplication in the rings $\mathcal{Q}$ and $\mathcal{R}_q$ from $\mathcal{O}(n^2)$ to $\mathcal{O}(n\log n)$, a negative wrapped convolution (NWC) method is applied with FFT and NTT. The NWC method chooses the set Ω_ϕ of roots ζ_k of $\phi = x^n + 1$ over $\mathbb{C}$ or $\mathbb{Z}_q$ as twiddle factors. Afterward, the expression of FFT and NTT is converted to equation (2) [5].

$$\hat{a}_k = \sum_{i=0}^{n-1} a_i \cdot \zeta_k^i = f(\zeta_k), \text{ with } \phi = x^n + 1 = \prod_{k=0}^{n-1} (x - \zeta_k) \quad (2)$$

It is to be noted that the modulus $q \equiv 1 \bmod 2n$ in FALCON, so the roots ζ_k exist over $\mathbb{Z}_q$. For FFT, the set of complex ζ_k is equation (3). It's easy to certificate that $f(\zeta_{n-1-k}) = \overline{f(\zeta_k)}$, so we only need to keep the first half of the complex numbers in FFT representation.

$$\Omega_\phi = \{\zeta_k = e^{\frac{j(2k+1)\pi}{n}} \mid 0 \le k < n\} \quad (3)$$

FALCON heavily relies on FFT and NTT to process polynomial operations. There are three vector formats for polynomial representation: coefficient format, FFT format, and NTT format. We can convert coefficient format to and from FFT or NTT format in $\mathcal{O}(n\log n)$ operations through FFT/IFFT (inverse FFT) or NTT/INTT (inverse NTT) algorithms. In FFT or NTT format, polynomial operations such as additions, subtractions, multiplications, and divisions can be performed in a point-wise manner. Therefore, polynomial multiplications and divisions can be conducted efficiently with FFT or NTT representations. In FALCON, verification is realized based on NTT. However, the signing procedure relies on both since fast Fourier sampling uses FFT, and private key calculation uses NTT.

Algorithm 1 Signature Generation

Input: A message m, a private key sk = $(\hat{\mathbf{B}}, \mathbf{T})$, a bound $\lfloor\beta^2\rfloor$
Output: A signature sig of m

r ← $\{0,1\}^{320}$ uniformly
c ← HashToPoint(r||m, q, n)
$\mathbf{t} \leftarrow \left(-\frac{1}{q}\mathsf{FFT}(c) \odot \mathsf{FFT}(F), \frac{1}{q}\mathsf{FFT}(c) \odot \mathsf{FFT}(f)\right)$
do
  do
    $\mathbf{z} \leftarrow \mathsf{ffSampling}_n(\mathbf{t}, \mathbf{T})$
    $\mathbf{s} \leftarrow (\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$
  while $\|\mathbf{s}\|^2 > \lfloor\beta^2\rfloor$
  (s_1, s_2) ← IFFT(s)
  s ← Compress(s_2, 8·sbytelen−328)
while s = $\perp$
return sig = (r, s)

Algorithm 2 Signature Verification

Input: A message m, a signature sig = (r, s), a public key pk = h, a bound $\lfloor\beta^2\rfloor$
Output: Accept or reject

c ← HashToPoint(r||m, q, n)
s_2 ← Decompress(s, 8·sbytelen−328)
if $s_2 = \perp$ **then**
  reject
$s_1 \leftarrow c - s_2 h \bmod q$
if $\|(s_1, s_2)\|^2 \le \lfloor\beta^2\rfloor$ **then**
  accept
else
  reject

C. FALCON Signing and Verifying Algorithms

Algorithm 1 describes the FALCON signing routine [5] that inputs a message m, a private key sk, an acceptance bound $\lfloor\beta^2\rfloor$ and outputs a signature sig. Firstly, HashToPoint calculates a hash polynomial c from a 320-bit nonce r and a message m with SHAKE-256. SHAKE-256 is one of the secure hash algorithm 3 (SHA-3) functions that NIST has standardized. FALCON relies on the SHAKE-256 hash function and some salt to encode the message m to a plaintext c.

Secondly, a pre-image $\mathbf{t}$ of c is computed to satisfy $\mathbf{t} \cdot \hat{\mathbf{B}} =$ (FFT(c),FFT(0)), where:

$$\hat{\mathbf{B}} = \left[\begin{array}{c|c} \mathsf{FFT}(g) & -\mathsf{FFT}(f) \\ \hline \mathsf{FFT}(G) & -\mathsf{FFT}(F) \end{array}\right] \quad (4)$$

Polynomials $f, g, F, G \in \mathcal{R}_q$ in equation (4) actually satisfy the NTRU equation:

$$fG - gF = q \bmod \phi \tag{5}$$

Therefore, line 3 in Algorithm 1 realizes fast calculation of $\mathbf{t}$ with FFT and point-wise multiplication $\odot$. Afterward, $\mathbf{t}$ and FALCON tree $\mathbf{T}$ are sent to a fast Fourier sampling procedure (ffSampling), which returns a vector $\mathbf{z}$ by randomized rounding to the vector $\mathbf{t}$. If the vector $\mathbf{s}$ obtained by $\mathbf{t}, \mathbf{z}, \hat{\mathbf{B}}$ in line 7 is short enough, two short polynomials s_1 and s_2 are returned through IFFT of $\mathbf{s}$. In the end, s_2 is encoded to a bitstring s according to the FALCON compression function (Compress). If s is not too long, a FALCON signature composed of r and s is generated. Through the above analysis, massive DLP can be explored in the signing procedure in line 3, line 6, line 7, and line 9. These DLP can be utilized for SIMD acceleration due to the large amount of polynomial operations with the degree $n = 512$ or 1024.

A discrete Gaussian sampling subroutine in ffSampling also constitutes the computational bottleneck. This kind of sampler requests support for variable floating-point mean and standard deviation. Moreover, floating-point function units require 64 bits of precision, which is unfriendly to constrained devices. A high-performance hardware accelerator of this discrete Gaussian sampling is needed for faster signing.

Algorithm 2 displays the FALCON verifying procedure [5], which has a few operations to determine whether a signature sig is valid. Firstly, like signing, the nonce r and the message m are concatenated and then hashed to a plaintext polynomial c. After obtaining s_2 by the FALCON decompression function (Decompress), we can calculate the value of s_1 in line 5. If Decompress returns the correct value of s_2 and polynomials s_1, s_2 are short enough, the signature sig is accepted. Therefore, faster SHAKE-256 in line 1, modular operations in line 5, and NTT are significant for FALCON verifying acceleration.

III. Proposed Design

A. Custom Instruction Extension

RVV provides a powerful model for data-parallel computing with flexible vector register mapping and convenient intrinsic API invoking. The selected element width (sew) and vector length multiplier ($lmul$) are defined to map vector elements to vector registers. The operand of a vector instruction is a register group composed of $lmul$ vector registers. Hence, the number of sew-bit elements in a group of $vlen$-bit vector registers is up to $vlmax = lmul \times vlen/sew$. To facilitate vectorized programming, RVV intrinsic[1] provides an interface to invoke RVV instructions in C/C++ code. With the flexibility of RVV and intrinsic, custom extensions based on RVV can take full advantage of RISC-V vectorized computing [11].

A vast instruction encoding space is specially reserved for custom instructions in RISC-V. The RISC-V custom extension shall fully employ the strength and comply with the regulation of RISC-V. Based on this customization mechanism, required instructions on the RISC-V scalar-vector basis can be designed for efficient applications. Therefore, this work proposes RVCE-FAL, a custom extension founded on the RISC-V scalar-vector architecture to accelerate FALCON, as shown in TABLE I.

Algorithm 3 Vectorized n-point In-Place NTT

Input: Polynomial $x \in \mathcal{R}_q$, precomputed ζ_{2n}^i for $i \in [0, n-1]$ stored in bit-reversed order of i in array zeta, the max number of elements in a vector register group $vlmax$
Output: $\hat{x} = \mathsf{NTT}(x)$

1: **for** $(m = 1; m \leq n/vlmax; m = 2m)$ **do**
2: $\quad d \leftarrow n/2m$
3: $\quad$ **if** $d \geq vlmax$ **then**
4: $\quad\quad$ **for** $(i = 0; i < m; i = i + 1)$ **do**
5: $\quad\quad\quad \omega \leftarrow \mathsf{zeta}[m + i]$
6: $\quad\quad\quad$ **for** $(j = 0; j < d/vlmax; j = j + 1)$ **do**
7: $\quad\quad\quad\quad$ **for** $(k = 0; k < vlmax; k = k + 1)$ **do**
8: $\quad\quad\quad\quad\quad idx \leftarrow i \cdot 2d + j \cdot vlmax + k$
9: $\quad\quad\quad\quad\quad u \leftarrow x[idx]$
10: $\quad\quad\quad\quad\quad v \leftarrow x[idx + d]$
11: $\quad\quad\quad\quad\quad x[idx] \leftarrow u + v \cdot w \bmod q$
12: $\quad\quad\quad\quad\quad x[idx + d] \leftarrow u - v \cdot w \bmod q$
13: $\quad\quad\quad\quad$ **end for**
14: $\quad\quad\quad$ **end for**
15: $\quad\quad$ **end for**
16: $\quad$ **else**
17: $\quad\quad$ **for** $(i = 0; i < m; i = i + 1)$ **do**
18: $\quad\quad\quad \hat{x} \leftarrow \mathsf{NTT}_{vlmax}(x, \mathsf{zeta}, n/vlmax, i)$
19: $\quad\quad$ **end for**
20: $\quad$ **end if**
21: **end for**
22: **return** $\hat{x}$

Our proposed custom instructions make the utmost of multiple types of register files in RISC-V scalar-vector architecture. These custom instructions in RVCE-FAL can efficiently execute FALCON-specific and general modular arithmetic operations, SHA-3 functions, Chacha20 calculation, and discrete Gaussian sampling. We develop powerful hardware, which constitutes a domain-specific architecture (DSA) core for faster FALCON. The implementation of SHA-3 functions[2] and Chacha20 calculation[3] have matured to be integrated into the overall system. Hence, the concentration of the following part will be put on the details of the modular arithmetic module and the discrete Gaussian hardware sampler.

B. Vectorized Computing for FALCON

According to the analysis in Section II-C, multiple polynomials in FFT format perform point-wise floating-point addition, subtraction, and multiplication with a high degree of parallelism in the FALCON signing procedure. We utilize RVV general floating-point instructions, such as vfadd.vv, vfsub.vv, vfmul.vv, vfmacc.vv, and so on, to exploit DLP for acceleration thoroughly. Besides, polynomial arithmetic over $\mathcal{R}_q$ exists in both the FALCON signing and verifying procedures. Polynomials in NTT format will perform point-wise modular arithmetic over

[1] https://github.com/riscv-non-isa/rvv-intrinsic-doc

[2] https://github.com/secworks/sha3

[3] https://github.com/utkarshb1/CHACHA20-Verilog-Code

TABLE I
BRIEF INTRODUCTION OF THE CUSTOM EXTENSION BASED ON THE RISC-V SCALAR-VECTOR FRAMEWORK

Instruction	Function[a]	Description
pqccsrrw	$rd \leftarrow \text{PqcCsr}[imm]$, $\text{PqcCsr}[imm] \leftarrow rs1$	Read and write a custom PQC control and status register (CSR)
vmodadd.vv	$vd[i] \leftarrow vs2[i] + vs1[i] \bmod q$	Perform modular addition between elements in vector register groups $vs2$ and $vs1$
vmodsub.vv	$vd[i] \leftarrow vs2[i] - vs1[i] \bmod q$	Perform modular subtraction between elements in vector register groups $vs2$ and $vs1$
vmodmul.vv	$vd[i] \leftarrow vs2[i] \times vs1[i] \bmod q$	Perform modular multiplication between elements in vector register groups $vs2$ and $vs1$
vmodmul.vx	$vd[i] \leftarrow vs2[i] \times rs1 \bmod q$	Perform modular multiplication between $rs1$ and elements in vector register group $vs2$
vmodq.x.xu.v	$vd[i] \leftarrow vs2[i] > q/2 \ ? \ vs2[i] - q : vs2[i]$	Convert elements in vector register group $vs2$ from $[0, q-1]$ to $[-q/2, q/2]$
vmodq.xu.x.v	$vd[i] \leftarrow vs2[i] < 0 \ ? \ vs2[i] + q : vs2[i]$	Convert elements in vector register group $vs2$ from $[-q/2, q/2]$ to $[0, q-1]$
vmod12289.v	$vd[i] \leftarrow vs2[i] \bmod 12289$	Perform specific modular reduction for 16-bit elements in vector register group $vs2$
inplacentt.vv	$vd \leftarrow \text{NTT}_{vlmax}(vs2, vs1)$	Perform small-scale NTT for elements in $vs2$ with twiddle factors in $vs1$
inplaceintt.vv	$vd \leftarrow \text{INTT}_{vlmax}(vs2, vs1)$	Perform small-scale INTT for elements in $vs2$ with twiddle factors in $vs1$
keccakload	$keccak[0] \leftarrow vs2$	Load elements in $vs2$ to the keccak data register $keccak[0]$
keccakstore	$vs2 \leftarrow keccak[imm]$	Store to elements in $vs2$ from the keccak data register $keccak[imm]$
keccakinit	no change to registers	Initialize the keccak state register in the SHA-3 module
keccakabsorb	$rd \leftarrow rs1 - sha3rate$	Absorb the input data in $keccak[0]$ into the Keccak state
keccaksqueeze	no change to registers	Squeeze from the keccak state into the output data in $keccak[1]$
chacha20init	no change to registers	Initialize the ChaCha20 state register in the ChaCha20 module
samplerz	$fd \leftarrow \text{samplerz}(fs2, fs1)$	Perform discrete Gaussian sampling with the mean $fs2$ and the standard deviation $fs1$

[a] rs, fs, and vs represent the source operands of integer registers, floating-point registers, and vector register groups, while rd, fd, and vd represent the destination operands. Moreover, $vd[i]$ refers to any i-th element in the register group vd, and imm denotes the immediate operand.

$\mathbb{Z}_q$. We utilize custom modular arithmetic instructions, such as vmodadd.vv, vmodsub.vv, vmodmul.vv, and so on, to achieve parallel speedup. According to Fermat's little theorem, modular division can transform into multiple modular multiplications. Thus, we do not customize the instruction for modular division.

NTT and FFT play a crucial role in FALCON signing and verifying. We propose a vectorized version for computing NTT as presented in Algorithm 3. ζ_{2n} is one of the $2n$-th primitive roots of unity, and $vlmax$ determines the number of modular operations executed in parallel by one instruction. Based on the Cooley-Tukey (CT) butterfly, this proposed algorithm requires no redundant operations, such as bit-reversing, pre-processing, and post-processing. Moreover, we can split arbitrary n-point NTT into vector modular operations (lines 10-11) and several parallel small-scale $vlmax$-point NTT_{vlmax} (line 18). Besides, this vectorized NTT algorithm supports the different radix, such as 2, 4, and 8. To demonstrate the improvement introduced by custom instructions, we implement radix-2 vectorized NTT in this work. Similarly, we also vectorize INTT relying on the Gentleman-Sande (GS) butterfly. Moreover, this work optimizes FFT and IFFT by this approach but has to execute small-scale FFT and IFFT on the scalar program due to unworthy hardware resource expenses.

C. Modular Arithmetic Hardware for FALCON

This paper designs a non-pipelined, modulus-general, 8-lane, and multi-function modular arithmetic accelerator, as illustrated in Figure 1. Considering the modulus $q = 12289$ in FALCON, elements over $\mathbb{Z}_q$ are 16-bit. Hence, this proposed design aims to support any modulus $q < 2^{16}$. One modular arithmetic lane consists of a 16-bit adder, a 16-bit subtracter, and a 16-bit multiplier. It realizes the Montgomery modular reduction (for vmodmul.v{v,x}), $[0, 5q)$ modular reduction (for vmod12289.v), and $[0, 2q)$ modular reduction (for vmodadd.vv and vmodsub.vv) based on the control of multiplexers and register files. Actually, fundamental arithmetic $(+, -, \times)$ circuits in this accelerator are

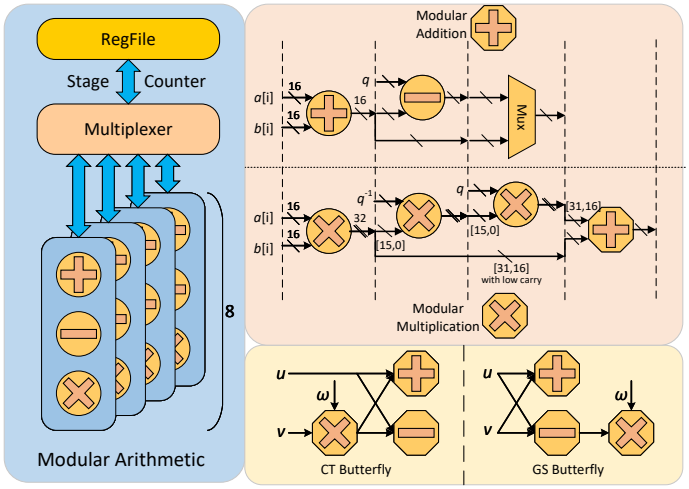

Fig. 1. The modular arithmetic hardware design.

highly reusable by different custom instructions. Distinctively, $[0, 5q)$ modular reduction is implemented through $x - \frac{x}{2^{14}} \times q$ without using multipliers. Due to specific $q = 2^{13} + 2^{12} + 1$, the $\times q$ operation converts to shifting and adding operations.

With $lmul = 1$, $vlen = 256$, and $sew = 16$ settings, a custom modular arithmetic instruction, except for inplacentt.vv and inplaceintt.vv, will use this modular arithmetic accelerator twice to obtain results due to $vlmax = 16$ elements but 8 lanes. To implement small-scale 16-point NTT and INTT, we reuse the modular addition, subtraction, and multiplication dataflows with a stage control register. Fortunately, 8 lanes are just enough for inplacentt.vv and inplaceintt.vv to execute stage by stage.

D. Discrete Gaussian Sampling Accelerator for FALCON

As displayed in Algorithm 4, the discrete Gaussian sampling procedure inputs floating-point mean μ and standard deviation σ and outputs a sampled integer z obeying discrete Gaussian distribution closely. It obtains an integer z_0 first from a half-

Algorithm 4 SamplerZ - Discrete Gaussian Sampling

Input: Floating-point values μ, σ' such that $\sigma' \in [\sigma_{min}, \sigma_{max}]$
Output: A sampled integer $z \in \mathbb{Z}$

1: $r \leftarrow \mu - \lfloor \mu \rfloor$
2: $ccs \leftarrow \sigma_{min}/\sigma'$
3: **while** (1) **do**
4: $\quad u \leftarrow$ UniformBits(72)
5: $\quad z_0 \leftarrow 0$
6: $\quad$ **for** $i = 0$ to 17 **do**
7: $\quad\quad$ **if** $u <$ RCDT$[i]$ **then**
8: $\quad\quad\quad z_0 \leftarrow z_0 + 1$
9: $\quad b \leftarrow$ UniformBits(8) & 0x1
10: $\quad z \leftarrow b + (2 \cdot b - 1) z_0$
11: $\quad x \leftarrow \frac{(z-r)^2}{2\sigma'^2} - \frac{z_0^2}{2\sigma_{max}^2}$
12: $\quad s \leftarrow \lfloor x / \ln(2) \rfloor$
13: $\quad d \leftarrow x - s \cdot \ln(2)$
14: $\quad s \leftarrow \min(s, 63)$
15: $\quad y \leftarrow$ C$[0]$
16: $\quad$ **for** $i = 0$ to 12 **do**
17: $\quad\quad y \leftarrow$ C$[i] - (d \cdot y)$
18: $\quad y \leftarrow \lfloor ccs \cdot y \cdot 2^{63} \rfloor$
19: $\quad v \leftarrow (2 \cdot y - 1) \gg s$
20: $\quad p \leftarrow 64$
21: $\quad$ **do**
22: $\quad\quad p \leftarrow p - 8$
23: $\quad\quad w \leftarrow$ UniformBits(8) $- ((v \gg p)$ & 0xFF)
24: $\quad$ **while** $((w = 0)$ and $(p > 0))$
25: $\quad$ **if** $w < 0$ **then**
26: $\quad\quad$ **return** $z + \lfloor \mu \rfloor$

Guassion distribution centered on 0 and a standard deviation of 1.8205 (lines 4-8). The RCDT array inside (line 7) is the reverse cumulative distribution table that has been pre-computed. Afterward, the SamplerZ method performs rejection sampling (lines 11-25) that calculates $ccs \cdot e^{-x}$ (lines 15-18) approximately and then decides whether to accept the sampled value $z + \lfloor \mu \rfloor$ (lines 19-25). The constant array C inside is pre-calculated and pre-stored. Except for the half-Guassion sampler, the floating-point function units are required in this SamplerZ method.

As shown in Figure 2, we propose the first full hardware accelerator for FALCON discrete Gaussian sampling. Our work utilizes the open-source Berkeley hardfloat[4] to implement high-precision floating-point arithmetic. Through the analysis for the target SamplerZ method, a 64-bit floating-point multiply-sub circuit ($c - a \times b$) is critical for acceleration, which is especially useful for $ccs \cdot e^{-x}$ function. Moreover, this fused multiply-sub circuit can execute floating-point multiplication or subtraction to save hardware resources. The conversion circuits between floating-point numbers and integer numbers are also necessary due to the need for $\lfloor \cdot \rfloor$ rounding down and operations between these two data types. However, the exquisite dataflow control is challenging for hardware design because of this sophisticated algorithm with frequent conversions between floating-point and integer. The proposal utilizes the state machine and multiplexers

[4]https://github.com/ucb-bar/berkeley-hardfloat

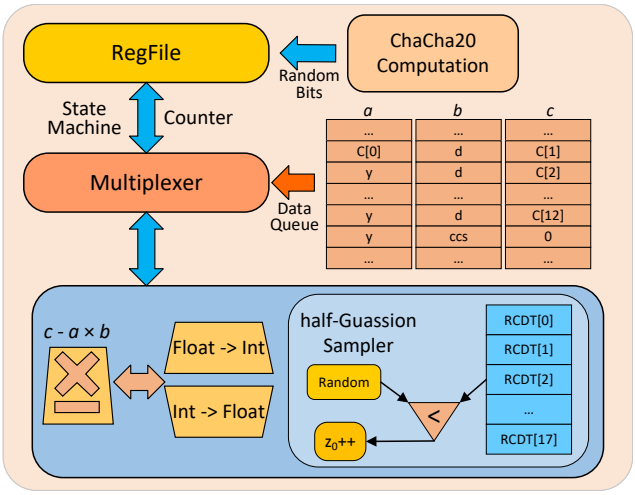

Fig. 2. The discrete Gaussian sampling hardware design.

to control data queue inputs of the fused floating-point multiply-sub circuit, forming a hardware discrete Gaussian sampler. To realize the UniformBits function, we call the ChaCha20 module to offer uniform random numbers for security reasons. It is to be noted that σ_{min} in FALCON-512 differs from that in FALCON-1024. Hence, the proposed design with the corresponding value σ_{min} is only suitable for FALCON-512 or FALCON-1024.

IV. Experimental Results

To improve flexibility, we implement the proposed custom hardware using Chisel HDL 3.6.0. We also evaluate the hardware design on the Xilinx UltraScale+ ZCU104 FPGA board using Xilinx Vivado 2019.2.

We build an in-house RTL simulation and validation platform with the gem5 simulator [12]. We design a software interface for the gem5 CPU model to transfer custom instructions and register files to the DSA core. This gem5-RTL co-simulation platform provides the simulation waveform and execution time of the proposed custom instructions running on the DSA core. Furthermore, we execute the optimized FALCON programs on an open-source RVV-enabled gem5 simulator[5]. For modeling the latency of RVV instructions, we adopt the actual latency of the commercial processor Xuantie C910 [13] with 4-lane vector function units and 256-bit vector register files. We utilize results on the gem5-RTL co-simulation platform to model the latency of custom instructions. The following experimental results rely on the above settings running on the gem5 MinorCPU model with the -O3 compile option[6]. We run the reference[7] software with the same MinorCPU model and the same compile option.

As shown in Table 1, our proposed sampler, accelerating the reference SW implementation 32×, outperforms the HW/SW implementation in [9] with a 16× improvement in cycle count. Moreover, the FPGA-based area consumption of the proposed sampler is close to the design in [9]. The floating-point fused multiply-sub circuit occupies over half of the resource usage.

[5]https://github.com/plctlab/plct-gem5
[6]https://github.com/riscv-collab/riscv-gnu-toolchain
[7]https://falcon-S.info/falcon-round3.zip

TABLE II
COMPARISON RESULTS OF DISCRETE GAUSSIAN SAMPLING AND NTT

Operation	Work	LUT/FF/DSP/BRAM	F_{max} (MHz)	Cycle Count
Discrete Gaussian Sampler	[9] HW/SW	3055/1960/9/0	117	1726
	Reference SW	-	-	3311
	This work HW	**4761/692/9/0 (2206/0/9/0)[a]**	**83**	**104**
$q = 12289$ $n = 1024$ NTT	[14] HW/SW	417/462/0/0	-	180237
	[15] HW/SW	886/618/26/1	-	24609
	Reference SW	-	-	152204
	This work HW/SW	**7839/416/8/0**	-	**11781**

[a] The usage of a floating-point fused multiply-sub circuit.

TABLE III
COMPARISON RESULTS OF FALCON DIGITAL SIGNATURE

Work	Procedure[a]	LUT/FF/DSP/BRAM	Latency $Kcycles$
[7] HW	FAL-512-V	57604/17764/26/18	77
	FAL-1024-V	58645/18241/28/18	161
[8] HW/SW	FAL-512-V	7219/3238/6/7	315
	FAL-1024-V		614
Reference SW	FAL-512-V	-	431
	FAL-1024-V		896
	FAL-512-S		28133
	FAL-1024-S		61302
This work HW/SW	FAL-512-V	**11454/851/8/0**	**62**
	FAL-1024-V		**130**
	FAL-512-S	**18565/3210/17/0[b]**	**1554**
	FAL-1024-S		**3188**

[a] FAL-512-V and FAL-512-S denote FALCON-512 verifying and signing routines respectively. And the same go FAL-1024-V and FAL-1024-S.
[b] We add the Gaussian sampler and the ChaCha20 module for signing.

Compared with other HW/SW implementations of NTT with $q = 12289$ and $n = 1024$, our design consumes more hardware usage to achieve higher speed and more functions. Based on the vectorized NTT algorithm, the proposed realization performs 12.9× and 2.1× better against the reference and the design in [15], respectively. Moreover, the proposed INTT with $n = 1024$ running with 13764 cycles accelerates the reference SW 14.5× and achieves 1.8× better than the work in [15]. We can even improve polynomial division in NTT format 26× for FALCON.

Table III shows the cycle count and area consumption results of our proposal. Our work consumes less LUT/FF/DSP/BRAM than the high-level synthesis (HLS) HW design of FALCON verification in [7]. Using HW/SW co-optimization, our proposal outperforms the state-of-the-art design [8] with a 5× speedup in cycle count. Compared with the reference SW, RVCE-FAL achieves a speedup for signing and verifying with 18× and 6.9× in FALCON-512 and 19× and 6.9× in FALCON-1024.

Actually, the reference software performs floating-point operations without floating-point function units (FPU) but integer function units for signing on constrained devices. FPU enables the signature generation with about 7.5× speedup. Our proposal achieves 2.5× better than the FPU-optimized FALCON signing. The RVV vector engine with built-in 4-lane FPU only performs 1.5× improvement in cycles with the vectorized FFT algorithm.

V. CONCLUSION

FALCON is one of the standardized digital signature algorithms selected by NIST. Moreover, NIST will release the draft standard for FALCON in 2024. However, hardware accelerators for complete FALCON signature processing routines have been omitted until now. In this paper, we propose an efficient custom extension founded on the RISC-V scalar-vector architecture to accelerate applications of FALCON for low-latency terminal devices in modern IoT. Our proposal accelerates the reference software implementation up to 18× and 6.9× in the FALCON signing and verifying procedures according to the results on the gem5-RTL simulation platform.

REFERENCES

[1] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134.
[2] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.
[3] T. Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, 1985.
[4] National Institute of Standards and Technology (NIST). Post-quantum cryptography selected algorithms 2022. [Online]. Available: https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022
[5] T. Prest, P. A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon: Fast-fourier lattice-based compact signatures over ntru," 2020. [Online]. Available: https://falcon-sign.info/falcon.pdf
[6] L. Beckwith, D. T. Nguyen, and K. Gaj, "Hardware accelerators for digital signature algorithms dilithium and falcon," *IEEE Design & Test*, 2023, early Access. [Online]. Available: https://doi.org/10.1109/MDAT.2023.3305156
[7] D. Soni, K. Basu, M. Nabeel, N. Aaraj, M. Manzano, and R. Karri, *Hardware architectures for post-quantum digital signature schemes*. Springer, 2021.
[8] P. Karl, J. Schupp, T. Fritzmann, and G. Sigl, "Post-quantum signatures on risc-v with hardware acceleration," *ACM Transactions on Embedded Computing Systems*, 2023, just Accepted. [Online]. Available: https://doi.org/10.1145/3579092
[9] E. Karabulut and A. Aysu, "A hardware-software co-design for the discrete gaussian sampling of falcon digital signature," Cryptology ePrint Archive, Paper 2023/908, 2023. [Online]. Available: https://eprint.iacr.org/2023/908
[10] A. Waterman et al., "Risc-v "V" vector extension version 1.1-draft," 2023. [Online]. Available: https://github.com/riscv/riscv-v-spec/releases/download/zvfh/zvfh-public-review-complete.pdf
[11] Y. Zhao, H. Kuang, Y. Sun, Z. Yang, C. Chen, J. Meng, and J. Han, "Enhancing risc-v vector extension for efficient application of post-quantum cryptography," in *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2023, pp. 10–17.
[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718
[13] C. Chen, X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen, C. Li, Y. Pu, J. Meng, X. Yan, Y. Xie, and X. Qi, "Xuantie-910: Innovating cloud and edge computing by risc-v," in *2020 IEEE Hot Chips 32 Symposium (HCS)*, 2020, pp. 1–19.
[14] E. Karabulut and A. Aysu, "Rantt: A risc-v architecture extension for the number theoretic transform," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, 2020, pp. 26–32.
[15] T. Fritzmann, U. Sharif, D. Müller-Gritschneder, C. Reinbrecht, U. Schlichtmann, and J. Sepulveda, "Towards reliable and secure post-quantum co-processors based on risc-v," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 1148–1153.