IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 1, pp. 203–226. DOI:10.46586/tches.v2025.i1.203-226

FalconSign: An Efficient and High-Throughput Hardware Architecture for Falcon Signature Generation

Yi Ouyang[1,+], Yihong Zhu[1,+], Wenping Zhu[1], Bohan Yang[1], Zirui Zhang[1], Hanning Wang[1], Qichao Tao[1], Min Zhu[2], Shaojun Wei[1] and Leibo Liu[1,*]

[1] Beijing National Research Center for Information Science and Technology (BNRist), School of Integrated Circuits, Tsinghua University, Beijing, China.
[2] Wuxi Micro Innovation Integrated Circuit Design Co., Ltd., Wuxi, China.
{oyy,bohanyang,wanghn,wsj,liulb}@tsinghua.edu.cn;{zhuyihong,zhuwp,qichaotao}@mail.tsinghua.edu.cn;zhangzir20@tsinghua.org.cn;zhumin@mucse.com
[+] These authors contributed equally to this work.
[*] Corresponding Author.

Abstract. Falcon is a lattice-based quantum-resistant digital signature scheme renowned for its high signature generation/verification speed and compact signature size. The scheme has been selected to be drafted in the third round of the post-quantum cryptography (PQC) standardization process due to its unique attributes and robust security features. Despite its strengths, there has been a lack of research on hardware acceleration, primarily due to its complex calculation flow and floating-point operations, which hinders its widespread adoption. To address this issue, we propose FalconSign, a high-performance, configurable crypto-processor designed to accelerate Falcon signature generation on FPGA/ASIC through algorithm-hardware co-design. Our approach involves a new scheduling flow and architecture for Fast-Fourier Sampling to enhance computing unit reuse and reduce processing time. Additionally, we introduce several optimized modules, including configurable randomness generation units, parallel floating-point processing units, and an optimized SamplerZ module, to improve execution efficiency. Furthermore, this paper presents a finely optimized hardware accelerator for the Falcon scheme. Our FPGA implementation results demonstrate a throughput improvement of approximately 5.1 × compared to state-of-the-art designs, with 2.8×/4.5×/4.2×/3.2× fewer in the area (LUTs/FFs/DSPs/BRAMs)-time product, for NIST security level V. The crypto-processor occupies an area of 0.71 mm^2 and achieves 5.2k OPS at throughput on the TSMC 28nm process for NIST security level I.

Keywords: Post-quantum cryptography · Falcon · Lattice · Fast-Fourier Sampling · Floating-point · High-performance · Configurable · FPGA

1 Introduction

In recent years, post-quantum cryptography (PQC) has garnered significant attention from both academia and industry. The development of quantum algorithms, such as Shor's [Sho94] and Grover's [Gro96] algorithms, has theoretically demonstrated the significant advantages of quantum computers in breaking traditional public-key cryptosystems like RSA and ECC. Recently, quantum computing has made considerable progress. For instance, in 2022, IBM unveiled its Osprey quantum computer, which boasts 433 qubits [PJSO24]. As a result of such advancements, it is widely believed that public-key cryptosystems will face severe security threats in the next 10 to 15 years [oST15]. To address this threat, the

Received: 2024-07-15 Accepted: 2024-09-15 Published: 2024-12-09

National Institute of Standards and Technology (NIST) initiated the PQC (post-quantum cryptography) project in 2016 to standardize PQC algorithms. In April 2022, NIST announced four PQC finalists: CRYSTALS-Dilithium [LDK+20], SPHINCS+ [BHK+19], Falcon [PFH+20], and Kyber [BDK+18]. Among these, Kyber is a key encapsulation algorithm, while the remaining three are digital signature algorithms. Following the establishment of these initial standards, NIST and the industry have been actively deploying and testing these four PQC algorithms.

Among these PQC schemes, Falcon stands out for its compact signature size, and quick verification process. Compared to the mainstream scheme, Dilithium, Falcon's verification speed and compact signature size are better, while its signature generation speed is slightly lower. This excellent performance makes Falcon irreplaceable in the post-quantum cryptography standardization process, particularly for time-sensitive applications like TLS (transport layer security) certificates [SKD20]. [TBRM22] stated that Falcon is the only viable scheme in the true hybrid design and vehicle-to-vehicle communication scenario. The latency and throughput of signature generation are crucial in various application scenarios, which include vehicle networking, settlement systems, and web servers. However, there is a lack of research on the performance boundary of Falcon signature generation, hindering its standardization and future application.

In fact, Falcon's signature generation process is more complex than its signature verification process, involving an intricate calculation flow and complex sampling operations. The recursive functions in Falcon constitute tree-like calculation-flow structures, adding to the complexity of hardware implementation. Furthermore, the utilization of double-precision floating-point operations sets Falcon apart from other PQC algorithms, which usually rely on integer computations. This unique characteristic introduces challenges regarding computational efficiency and resource utilization, as floating-point calculations typically require more time and storage. Collectively, these factors make implementing Falcon in hardware a challenging task. Consequently, there is a need to address the challenges mentioned above by optimizing floating-point computations, managing complex control flow structures, and efficiently utilizing hardware resources to ensure a balance between performance and hardware constraints.

Related Works. Karabulut et al. [KA24] proposed a discrete Gaussian sampler with a hardware/software co-design approach to improve sampler speed. On the Zynq-7000 SoC platform, this approach achieved a 9.83× improvement in sampling operations and a 2.7× overall improvement in the signature scheme compared to a software solution. However, they only designed a multi-stage pipelined multiplier with inputs of 73-bit and 69-bit signed operands to accelerate the multiplication and subtraction operations in sampling, without addressing the floating-point calculation bottlenecks outside of sampling. Lee et al. [LYN+24] also reported a hardware-software co-design implementation, reducing the cycle amount by 3.58× from the pure software implementation. The hardware platform used in their work was based on the 28 nm Samsung process, allowing only synthesis and place-and-route results. Yu et al. [YSZ+24] proposed a RISC-V scalar-vector custom extension for Falcon. The sampling module of this architecture does not operate at a high frequency, reaching only 83MHz. Additionally, it does not address the floating-point calculation bottlenecks outside of sampling. A recent study [SAW+23] utilized a HLS(high-level synthesis) based approach to directly map official codes to hardware design, without considering further optimizations of hardware architecture. However, the signature generation throughput is limited due to the lack of fine-grained optimizations, which limits exploration of the performance boundary of Falcon scheme.

Our Contributions. In this paper, we present an efficient and high-throughput hardware architecture for Falcon signature generation, and we will release the source code on `https://github.com/YiOuyang1/FalconSign`. In general, our contributions are summarized as follows:

1. A novel scheduling flow and architecture for `ffSampling` is proposed, which involves a dedicated State-Transition-Based mechanism. The data flow in `ffSampling` process is analyzed, and a compact control mechanism of the data flow within the tree structure is adopted to address Falcon's complex and recursive control overheads. This approach allows for dynamic updating of the tasks, resulting in minimal control overheads.

2. A compact and configurable FPU (Floating-point Processing Unit) is proposed. A memory-centric vectorized architecture and corresponding parallel algorithms for FFT/IFFT are selected due to their adaptability to the Falcon scheme. The principle of this structure and algorithm are demonstrated for the first time in this paper. Additionally, a conversion relation and parallel algorithm is provided between `splitfft/mergefft` operations and FFT/IFFT operations, which can reuse the same parallel architecture.

3. A pipelined module for the Gaussian sampler and other optimized modules are meticulously designed. We observe that adjacent discrete Gaussian samplings in Falcon use the same standard deviation, and thus part of the sampling computation time can be hidden. With careful pipeline design and an efficient random number generator, we maximized the parallelism of the SamplerZ circuit, achieving the lowest sampling latency known to date.

4. Besides the contributions listed above, several design trade-offs and optimizations are adopted. An efficient accelerator for Falcon signature generation is achieved. As a result, the final performance is 5.1× better than the state-of-the-art designs for NIST security level V. Further the area-time product (ATP) for LUTs/FFs/DSPs/BRAMs is improved by 2.8×, 4.5×, 4.2×, and 3.2×, respectively.

The rest of this paper is structured as follows: Section 2 introduces the notations used throughout this work and provides a mathematical background, including a summary of the Falcon protocol. Next, section 3 describes the architecture of the proposed accelerator as well as the data flow and tree-like `ffSampling` method. Section 4 discusses the designs of optimized modules involved in this paper. Section 5 presents the performance results on FPGA/ASIC and provides comparisons with related works. Finally, section 6 concludes the paper.

2 Preliminaries

2.1 Notations

$\mathbb{Z}_q$ denotes the quotient ring $\mathbb{Z}/q\mathbb{Z}$, for $q \in \mathbb{N}$. In Falcon, the modulus $q = 12289$ is prime, so $\mathbb{Z}_q$ is a finite field. We define the number fields as $\mathbb{Q}[x]/(\phi)$, where $\phi = x^n + 1$ for $n = 2^\kappa$. Matrices are represented using bold uppercase letters,(e.g. $\mathbf{B}$); vectors are denoted using bold lowercase letters, (e.g. $\mathbf{v}$); and scalars, which include polynomials, are presented in italic, (e.g. s). Let $\langle \cdot, \cdot \rangle$ denote the inner product, and $\| \cdot \|$ denote its associated norm. For polynomials represented by a and b, their inner product can be expressed as $\langle a, b \rangle = \frac{1}{\deg(\phi)} \sum_{\phi(\zeta)=0} a(\zeta) \cdot \overline{b(\zeta)}$. For a, its norm can be expressed as $\|a\| = \sqrt{\langle a, a \rangle}$. Extend the definitions of the inner product and norm to the vector dimensions. For $u = (u_i)_i$ and $v = (v_i)_i$ in $\mathbb{Q}^m$, the inner product $\langle u, v \rangle$ is defined as $\sum_i \langle u_i, v_i \rangle$. For our choice of ϕ, the inner product coincides with the usual coefficient-wise inner product: $\langle a, b \rangle = \sum_{0 \le i < n} a_i b_i$.

The discrete Gaussian distribution over integers can be represented as $D_{\mathbb{Z},\sigma,\mu}$, with parameters σ and μ. The distribution samples x on the set of integers $\mathbb{Z}$ with probability $D_{\mathbb{Z},\sigma,\mu}(x) = \frac{\rho_{\sigma,\mu}(x)}{\sum_{z\in\mathbb{Z}}\rho_{\sigma,\mu}(z)}$, where the Gaussian function is given by $\rho_{\sigma,\mu}(x) = \exp\left(-\frac{|x-\mu|^2}{2\sigma^2}\right)$.

2.2 Falcon scheme

Falcon is a lattice-based cryptographic scheme, abbreviated from "Fast Fourier lattice-based compact signatures over NTRU." It is based on the GPV theoretical framework [GPV08] for creating a hash-and-sign scheme, which requires two critical components: a class of cryptographic lattices and a trapdoor sampler. Falcon employs NTRU lattices [HPS98] and `ffSampling` [DP16] for these purposes. Therefore, Falcon can be succinctly described as follows: GPV framework + NTRU lattice + Fast Fourier Sampling. Regarding security in both classical and quantum computational models, the GPV framework has been proven secure under the SIS (Short Integer Solution problem) assumption [BDF+11].

As the high-level framework of Falcon, GPV uses a full-rank matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ (with $m > n$) as the public key and a matrix $\mathbf{B} \in \mathbb{Z}_q^{m\times m}$ as the private key. The public key and private key are orthogonal: $\mathbf{B} \times \mathbf{A}^t = 0 \mod q$. Given a message m, a signature of m is a short vector $s \in \mathbb{Z}_q^m$ such that $s\mathbf{A}^t = H(m)$, where $H : \{0,1\}^* \to \mathbb{Z}_q^n$ is a hash function. To verify a signature s given the public key $\mathbf{A}$, one must ensure that s is short and satisfies $s\mathbf{A}^t = H(m)$. This signature verification process is straightforward, but the key generation and signature generation processes are more delicate. Next, we will provide a detailed explanation of these three algorithms.

Falcon instantiates the GPV framework on the NTRU lattice. The core of the private key comprises four polynomials $f, g, F, G \in \mathbb{Z}[x]/(\phi)$ with short integer coefficients, which satisfy the NTRU equation:

$$fG - gF = q \mod (\phi) \tag{1}$$

The public key pk corresponding to the private key sk $= (f, g, F, G)$ is a polynomial $h \in \mathbb{Z}_q[x]/(\phi)$ such that $h = gf^{-1}\ mod\ (\phi, q)$.The key generation process consists of two steps: `NTRUGen` and Falcon Tree Compute. `NTRUGen` primarily generates the polynomials f, g, F, G that satisfy the NTRU equation. After obtaining f and g, it is necessary to efficiently solve the NTRU equation to obtain F and G.

The final part of key generation involves processing the sk into a suitable format that allows for fast signature generation. The Falcon tree provides this appropriate format. The generation of the Falcon tree is achieved by computing the LDL* decomposition of the matrix $G = \mathbf{B}\mathbf{B}^*$. Recursively use LDL* decomposition until the matrix size is 1, and then perform normalization. Structurally, the Falcon tree is essentially a binary tree.

The Falcon signing algorithm first computes the hash value of the message m, and the message m and a random value r are hashed to produce $c \in \mathbb{Z}_q[x]/(\phi)$. Then, using information from the key, two short values, s_1 and s_2, are generated, which satisfy the equation $s_1 + s_2h = c \mod q$. To securely generate s_1 and s_2, the algorithm uses a trapdoor sampler based on the FFT(fast Fourier transform), which relies on a precomputed Falcon tree. By recursively applying split and merge operations, the algorithm efficiently generates signatures while ensuring the private key remains secure. The sampler first generates $\mathbf{t} \leftarrow (c, 0)\cdot\mathbf{B}^{-1}$ and outputs $(s_1, s_2) \leftarrow (\mathbf{t} - \mathbf{z})\cdot\mathbf{B}$. Finally, s_1 and s_2 is compared to the bound to ensure it is short enough. The output signature consists of the salt r and the compressed form of s_2. The details of the signature generation can be found in Algorithm 1.

The signature verification process is much simpler than the key generation and signature generation processes. Given the public key pk $= h$, a message m, a signature sig $= (r, s)$, and an acceptance bound $\lfloor\beta^2\rfloor$, the verifier uses the public key pk to check whether sig is a valid signature of the message m.

Algorithm 1 FALCON.Sign(m, sk, $\lfloor \beta^2 \rfloor$) from [PFH+20]

```
Require: Message $m$, Secret key sk, A bound $\lfloor \beta^2 \rfloor$
Ensure: A signature sig of $m$
 1: $r \leftarrow \{0,1\}^{320}$ uniformly
 2: $c \leftarrow \texttt{HashToPoint}(r \parallel m, q, n)$
 3: $\mathbf{t} \leftarrow \left(-\frac{1}{q}\mathrm{FFT}(c) \odot \mathrm{FFT}(F), \frac{1}{q}\mathrm{FFT}(c) \odot \mathrm{FFT}(f)\right)$    ▷ $\mathbf{t} = (\mathrm{FFT}(c), \mathrm{FFT}(0)) \cdot \hat{\mathbf{B}}^{-1}$
 4: repeat
 5:     repeat
 6:         $\mathbf{z} \leftarrow \texttt{ffSampling}_n(\mathbf{t}, \mathbf{T})$
 7:         $\mathbf{s} = (\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$    ▷ $\mathbf{s}$ follows a Gaussian distribution: $\mathbf{s} \sim \mathcal{D}_{(c,0)+\Lambda(\mathbf{B}),\sigma,0}$
 8:     until $\|\mathbf{s}\|^2 \leq \lfloor \beta^2 \rfloor$    ▷ Compute $\|\mathbf{s}\|^2$ in FFT representation
 9:     $(s_1, s_2) \leftarrow \mathrm{invFFT}(\mathbf{s})$
10:     $\mathrm{s} \leftarrow \texttt{Compress}(s_2, 8 \cdot \mathrm{sbytelength} - 328)$
11: until $\mathrm{s} \neq \perp$
12: sig $\leftarrow (r, \mathrm{s})$
13: return sig
```

Parameter settings of the algorithm are shown in Table 1.Falcon provides signature generation schemes for two security levels: I and V. For security level I, the degree of the base polynomial is 512, and for security level V, the degree of the base polynomial is 1024. Both use a modulus of 12289, but the rejection threshold for signatures is different.

Table 1: Parameters of the two Falcon security levels.

	Falcon-512	**Falcon-1024**
Target NIST Level	I	V
Ring degree n	512	1024
Modulus q	12289	12289
Standard deviation σ	165.736617183	168.388571447
$\sigma_{\min}$	1.277833697	1.298280334
$\sigma_{\max}$	1.8205	1.8205
Max. signature square norm $\lfloor \beta^2 \rfloor$	34,034,726	70,265,242
Public key bytelength	897	1,793
Signature bytelength sbytelength	666	1,280

2.3 Fast Fourier Sampler

This section further explains the `ffSampling` (Fast Fourier Sampling) operation used in signature generation, which is a key focus of this work. Trapdoor sampling takes as input a matrix $\mathbf{A}$, a trapdoor T, and a target $\mathbf{c}$, and outputs a short vector $\mathbf{s}$ such that $\mathbf{s}^t\mathbf{A} = \mathbf{c}\, mod\, q$. The entire operation is performed in the FFT domain to improve computational efficiency. The `ffSampling` operation recursively uses split and merge steps, performing discrete Gaussian sampling at the lowest level to obtain $(\mathbf{s}_1, \mathbf{s}_2)$. This specific steps can be found in Algorithm 2.

Let ϕ and ϕ' be power-of-two cyclotomic polynomials such that $\phi(x) = \phi'(x^2)$. For example, $\phi(x) = x^n + 1$ and $\phi'(x) = x^{n/2} + 1$. The core of the algorithm requires the ability to split elements of $\mathbb{Q}[x]/(\phi)$ into two smaller elements in $\mathbb{Q}[x]/(\phi')$. Conversely, we also need the capability to merge two elements from $\mathbb{Q}[x]/(\phi')$ into one element in $\mathbb{Q}[x]/(\phi)$.The split and merge operations are also performed in the FFT domain, specifically using the `splitfft` and `mergefft` operations.

Algorithm 2 `ffSampling`$_n(t, T)$ from [PFH+20]

```
Require: $\mathbf{t} = (t_0, t_1) \in \mathrm{FFT}\left(\mathbb{Q}[x]/(x^n+1)\right)^2$, a Falcon tree $T$
Ensure: $\mathbf{z} = (z_0, z_1) \in \mathrm{FFT}\left(\mathbb{Z}[x]/(x^n+1)\right)^2$
 1: if $n = 1$ then
 2:     $\sigma' \leftarrow T$.value        ▷ It is always the case that $\sigma' \in [\sigma_{\min}, \sigma_{\max}]$
 3:     $z_0 \leftarrow$ SamplerZ$(t_0, \sigma')$
 4:     $z_1 \leftarrow$ SamplerZ$(t_1, \sigma')$
 5:     return $z = (z_0, z_1)$   ▷ Since $n = 1$, $t_i = \mathrm{invFFT}(t_i) \in \mathbb{Q}$ and $z_i = \mathrm{invFFT}(z_i) \in \mathbb{Z}$
 6: end if
 7: $(\ell, T_0, T_1) \leftarrow$ (T.value, T.leftchild, T.rightchild)
 8: $\mathbf{t_1} \leftarrow$ splitfft$(t_1)$        ▷ $t_0, t_1 \in \mathrm{FFT}\left(\mathbb{Q}[x]/(x^{n/2}+1)\right)^2$
 9: $\mathbf{z_1} \leftarrow$ ffSampling$_{n/2}(\mathbf{t_1}, T_1)$        ▷ First recursive call to ffSampling$_{n/2}$
10: $z_1 \leftarrow$ mergefft$(\mathbf{z_1})$        ▷ $z_0, z_1 \in \mathrm{FFT}\left(\mathbb{Z}[x]/(x^{n/2}+1)\right)^2$
11: $t_0' \leftarrow t_0 + (t_1 - z_1) \odot \ell$
12: $\mathbf{t_0} \leftarrow$ splitfft$(t_0')$
13: $\mathbf{z_0} \leftarrow$ ffSampling$_{n/2}(t_0, T_0)$        ▷ Second recursive call to ffSampling$_{n/2}$
14: $z_0 \leftarrow$ mergefft$(\mathbf{z_0})$
15: return $\mathbf{z} = (z_0, z_1)$
```

The second part of the sampler is the discrete Gaussian sampling function. This function was briefly introduced in 2.1. This section further details how to securely sample Gaussian samples $z \leftarrow D_{\mathbb{Z},\sigma',\mu}$ for any $\sigma' \in [\sigma_{\min}, \sigma_{\max}]$ and $\mu \in \mathbb{R}$. This is also a key focus in the computational process of signature generation. The sampling process is carried out by the `SamplerZ` function. This algorithm uses `BaseSampler` and `BerExp` as subroutines. `BaseSampler` samples an integer $z_0 \in \mathbb{Z}^+$ according to the distribution χ defined on $\{0, \ldots, 18\}$: $\forall i \in \{0, \ldots, 18\}, \chi(i) = 2^{-72} \cdot \mathrm{pdt}[i]$. The distribution χ is very close to the "half-Gaussian" $D_{\mathbb{Z}^+,\sigma_{\max}}$ in terms of Rényi divergence, with $R_{513}(\chi \parallel D_{\mathbb{Z}^+,\sigma_{\max}}) \leq 1+2^{-78}$. The `BerExp` subroutine performs rejection sampling to approximate $ccs \times exp(-x)$. This operation also requires the `BerExp` subroutine to call another subroutine, `ApproxExp` , to compute $exp(-x)$, completing the calculation of the entire rejection probability.

3 Design Decisions

In this section, the high-level hardware architecture for Falcon signature generation is presented. The computations in Falcon signature generation are categorized as tasks, and the execution logic of these tasks is described in this section. The control flow for complex recursive operations and storage design are also involved.

3.1 Memory-Centric Overall Architecture

The current architecture for lattice-based PQC algorithms can be divided into two categories. One approach involves cascading different functional modules along the data path [ZZW+22]. With meticulous design, this approach can achieve significant acceleration but often lacks flexibility and sufficient support for complex control patterns. Another approach is the instruction-set co-processor architecture, which offers better flexibility and ease of integration [RB20]. However, this architecture's simple instruction memory design still falls short in supporting complex controls, such as loop unrolling for recursion, leading to excessive resource consumption in Falcon implementations. Moreover, in cases where there are significant differences in the utilization rates of functional modules within the data path, some modules may not be fully utilized. Therefore, achieving high-performance

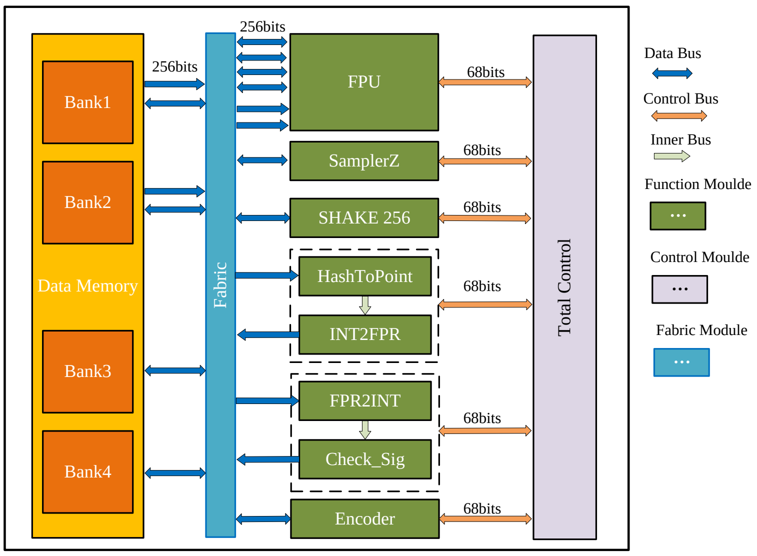


Figure 1: High-level architecture of FalconSign.

implementations requires careful consideration of data path balancing. Combining the strengths of both approaches, we propose a highly efficient memory-centric architecture for Falcon signature generation, as illustrated in Figure 1.

The whole architecture mainly consists of three main parts: Data Memory, Calculating Modules, and Total Control Unit. The Calculating Modules include a Floating-Point Processing Unit (FPU), Sample Unit (SamplerZ), Hash Module (SHAKE256), Message Sampling Module(HashToPoint), Floating Point to Fixed Point Conversion Modules (FPR2INT and INT2FPR), Signature Check Module (Check_Sig) and Encode Module(Encoder). The data width of the hardware architecture is 256 bits, and the control path width is 68 bits. The Data Memory is internally divided into four independently one-read one-write banks to meet the requirements of simultaneous read and write operations. The BRAM resources consumed by Data Memory vary with the security level. For Falcon-512 (security level-I), it consumes 45 BRAMs, while for Falcon-1024 (security level-V), it consumes 58 BRAMs.

The FPU handles all floating-point computations in the algorithm. The floating-point arithmetic operations are intensive in signature generation and result in a performance bottleneck. More importantly, high flexibility is required due to the complex calculation patterns in Falcon. The complex patterns include decomposition of the fast Fourier sampler and merging logic in Falcon, which requires floating-point arithmetic operations to support multiple vector sizes ($n = 1, 2, 4, 8, \dots, 256$). Therefore, we designed high-speed and configurable FPUs, as shown in Section 4.2.1.

The Sampling Unit (SamplerZ) implements the discrete Gaussian sampling algorithm, requiring 1024 or 2048 sampling operations for signature generation. The complex data structures and computational flow in SamplerZ, along with its frequent invocation, make it a significant bottleneck. Accordingly, the optimized and refined acceleration strategies are devised for them, as detailed in Section 4.3.

The SHAKE256, HashToPoint, INT2FPR, FPR2INT, Check_Sig, and Encoder modules are other essential functional modules in signature generation. The SHAKE256 module transfers the message m and salt r to generate a huge amount of random numbers. The HashToPoint module then uses these random numbers for rejection sampling to obtain the polynomial $c \in \mathbb{Z}_q^N$. The FPR2INT and INT2FPR modules convert between double-precision floating-point numbers and fixed-point numbers. Check_Sig module calculates the norm of the obtained signature s_2 and compares it with the threshold to determine

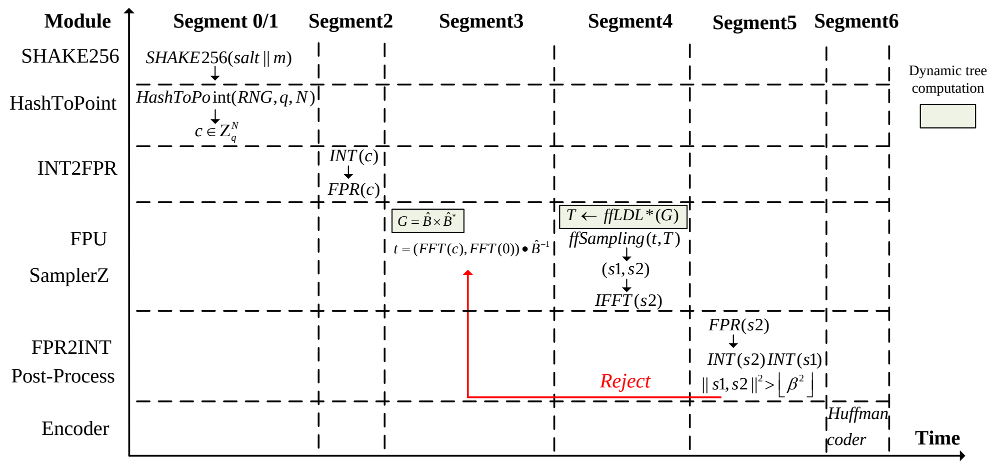


Figure 2: Task flow for signature generation (Dynamic tree computation refers to the additional computations required by Sign_Dynamic).

whether the signature is valid. Finally, the Encode module generates the valid signature. The utilization rates of these modules are limited. In most cases, each module is only utilized once during each attempt of signature generation. To balance their utilization of the data path and reduce fabric resource consumption, the HashToPoint unit and INT2FPR module are cascaded to fulfill pre-processing operations before sampling, and FPR2INT and the Check_Sig module are also cascaded for post-processing operations after sampling.

3.2 Dynamic Task Processing

Falcon provides two signature generation modes: Sign_Tree and Sign_Dynamic [PFH+20]. Sign_Tree uses a pre-expanded private key format called as “Falcon tree”, enabling faster signature generation. In contrast, Sign_Dynamic does not need to pre-compute the “Falcon tree”; instead, the “Falcon tree” is dynamically constructed during the signing process. FalconSign supports both methods on the same hardware using a dynamic task process mechanism. The major difference between these two methods lies in the expansion of FPU functionality (floating-point square root and division operations) and the adaptation of the control flow. The calculation of the “Falcon tree” need be integrated into the `ffSampling` process.

In the proposed architecture, the Total Control module generates the 68-bit tasks including the source address, destination address, and execution modes. Benefited from this pre-defined task format, the computation units can be scheduled by the proposed scheduling mechanism. The task flows for Sign_Tree and Sign_Dynamic in the FalconSign architecture are illustrated in Figure 2.

In Figure 2, the calculation of `ffSampling` and `ffLDL` construct the main challenge for control flow. The Falcon algorithm specification [PFH+20] describes the detailed procedures of `ffSampling` and `ffLDL`. These two procedures utilize recursive structures, involving a binary tree structure for the computation flow, which is similarly illustrated in [SAW+23, LZS+24]. During signature generation, every computational node in the binary tree need to be traversed in the specific order as shown in Figure 3. The depth of the binary tree is 9/10 in Falcon 512/1024, respectively.

For the control flow implementation of `ffSampling` and `ffLDL`, we propose a dynamic task updating strategy. This technique aims to reduce the resource consumption for task maintenance while enhancing the flexibility of task scheduling. First, we use three

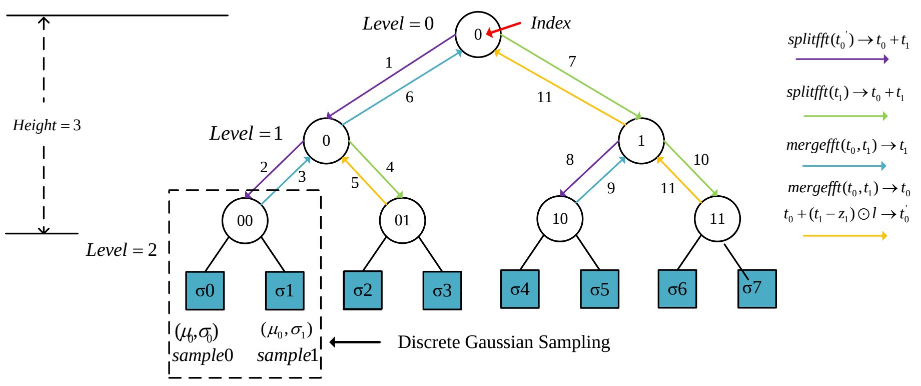

Height = 3
Level = 0
Level = 1
Level = 2
Index
0
00
01
10
11
1
2
3
4
5
6
7
8
9
10
11
σ0
σ1
σ2
σ3
σ4
σ5
σ6
σ7
(μ_0,σ_0) sample0
(μ_0,σ_1) sample1
Discrete Gaussian Sampling
$splitfft(t_0^{'}) \rightarrow t_0 + t_1$
$splitfft(t_1) \rightarrow t_0 + t_1$
$mergefft(t_0,t_1) \rightarrow t_1$
$mergefft(t_0,t_1) \rightarrow t_0$
$t_0 + (t_1 - z_1) \odot l \rightarrow t_0^{'}$

Figure 3: The scheduling mechanism of `ffSampling`. The index in the solid circles at each level represents the position index in the the layer. Dashed-line boxes indicate a shared μ used for two discrete Gaussian samplings.

coordinates $Level, Index$, and $State$ to track the position of the computation flow, as illustrated in the Figure 3. The $State$ is divided into RIGHT-DOWN, RIGHT-UP, LEFT-DOWN, and LEFT-UP, where each state corresponds to a sequence of operations. Second, the current task is dynamically generated using the coordinate information and then dispatched to the corresponding execution module, as shown in the Figure 4 and Equation 2. Due to the similarity in computations at each node, the dynamic update incurs minimal maintenance overhead. Finally, similar tasks at the same node are merged, such as the SamplerZ tasks shown in the Figure 3, reducing the number of task dispatches.

$$(tasks, State\_next) = \text{Task\_Update}(Level, Index, State) \tag{2}$$

Compared to the pre-filled program memory approach [RB20] , the dynamic task updating strategy reduces the storage overhead of the `ffSampling` tasks in Falcon 512/1024 from 36kB/72kB to 80/80 slices. Additionally, this strategy can support multiple sampling depths flexibly in `ffSampling` with minimal cost. For instance, adapting the task updating method from Falcon-512 to Falcon-1024 only requires changing register values that denotes the depth of the binary tree. Different from the approach presented in [SAW+23, LZS+24], a detailed hardware task execution mechanism is proposed for `ffSampling`. By merging similar tasks (e.g., two SamplerZ tasks with the same expected value), the number of sampling tasks is reduced by 256/512 in Falcon 512/1024, respectively.

4 Optimized Modules

This section introduces several optimized modules designed to achieve high-speed performance with minimal resource usage.

4.1 Compact Data Pre-processing Module

As shown in Algorithm 1, the input message is hashed through the SHAKE256 function [SD3] in the Falcon scheme. It only involves SHAKE256 once to generate a sufficient number of random numbers, followed by threshold q rejection sampling to produce the polynomial $c \in \mathbb{Z}_q^n$. Overall, the proportion of Keccak calculations and rejection sampling time is less than 1% of the total time. Thus, these modules are designed for low resource consumption and implementation complexity.

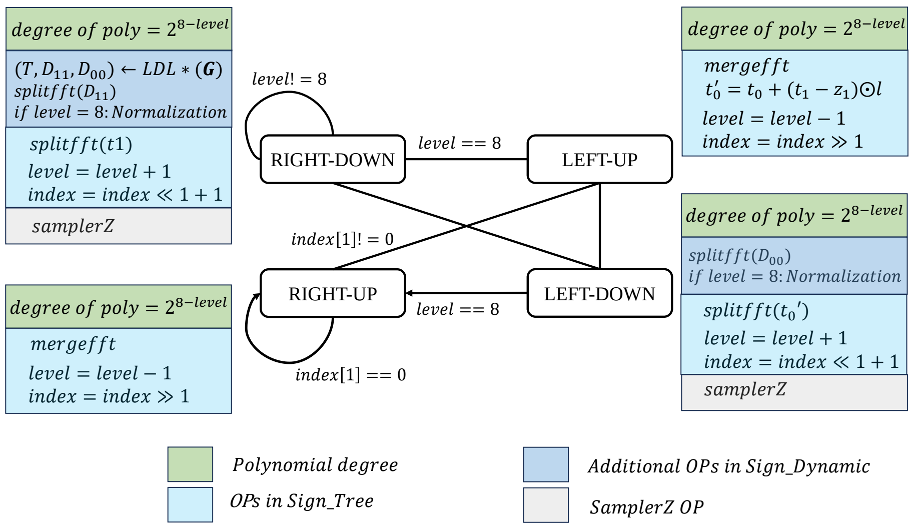


Figure 4: Task update strategy in `ffSampling` and `ffLDL`.

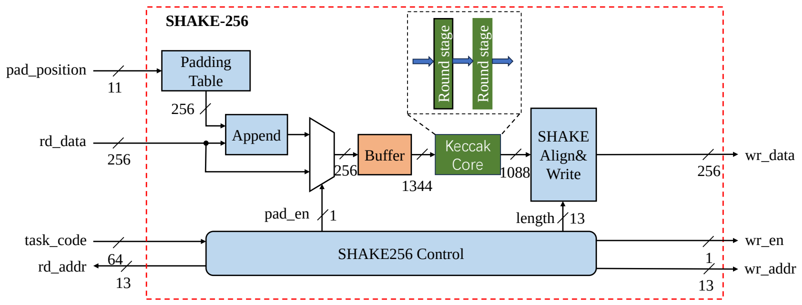


Figure 5: Efficient architecture of two-round SHAKE256.

A Keccak module is instantiated to perform the SHAKE256 function, which is shown in Figure 5. This two-round Keccak core requires 12 cycles to compute the "state-permutate" operation. After completing the absorbing phase of SHAKE256, it takes 12 cycles to output 1088 bits of data. The generated data is written into BRAM after alignment operations. For the rejection sampling operation, the algorithm takes a 16-bit random digit r each time. If r is less than kq (where $k = \lfloor 2^{16}/q \rfloor$), the value is accepted, and $c_i = r \mod q$ is obtained. The final sampled output is $c \in \mathbb{Z}_q^n$ (with $n = 512$ or 1024). The parallelism of the `HashToPoint` function is 4-coefficient (256-bit), which matches the data path of our architecture.

4.2 Vectorized Floating-Point Processing Unit

Dense floating-point computations occupy about 50% of the signature generation time [ZZO+24, KA24]and they become the bottlenecks of `ffSampling`. The acceleration of the floating-point computations requires a high-speed, low-latency, reconfigurable FPU. The low-latency requirement arises from the severe data dependencies between floating-point vector calculations, where deeply pipelined design would cause a significant number of pipeline bubbles. The reconfigurable requirement is dictated by the interconnections between the operators. The floating-point operators include complex-number addition, subtraction, multiplication, FFT/IFFT, `splitfft`, and `mergefft`. Among these, FFT/IFFT

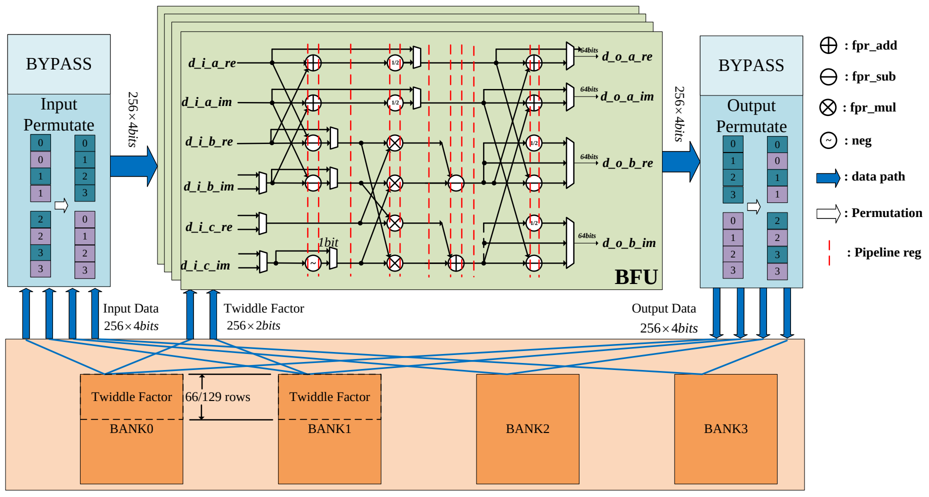


Figure 6: FFT/IFFT Architecture.

is only called three times, but its underlying BFU (butterfly computation unit) contains a rich set of floating-point add/sub/mul computational resources. Moreover, `mergefft` and `splitfft` are sub-processes of FFT/IFFT [PFH+20]. These operators are adapted to the FFT/IFFT architecture in our work, increasing computational resource utilization.

For the implementation of FFT, there are two hardware methodologies: pipelined FFT HW architectures [Gar22] and vectorized FFT HW architecture. The former architecture has problems with poor reconfigurability, high latency, and low compatibility in the Falcon signature scenario. Cascaded BFUs are not suitable for reconfiguration into vector computations. Secondly, the pipelined FFT architecture uses a large number of registers, which reduces memory access in FFT/IFFT operations. More importantly, the pipelined FFT architecture is not compatible with the `splitfft` and `mergefft` calculations, because these calculations need to be calculated separately by stage. However, in other floating-point computations, these registers are underutilized, resulting in poor area efficiency. The vectorized FFT architecture has hardware-friendly reconfigurability and low latency. However, traditional in-place calculations require extensive input-output permutation networks and intensive storage access. Considering the large BRAM consumption of the Falcon private key and the design goals of the operator, the vectorized FFT architecture is chosen in our architecture. Inspired by [ZZL+22]'s vectorized NTT architecture, a vectorized FFT architecture is proposed with a scalable parallel computing scheme for FFT/IFFT and `splitfft`/`mergefft`.

4.2.1 Vectorized FFT/IFFT with Conflict-Free Strategy

[XHY+20] introduced an in-place DIT-NTT computation hardware architecture, but it incurs substantial permutation networks in vectorized architectures. This is because the two operands of butterfly computation are scattered within a single vector, requiring permutation operations. The design of FFT has a similar issue. To address this, an FFT parallel scheme is proposed with a vector dimension of $l (l = 2^m, m \in \mathbb{N})$, which requires only one type of permutation network.

Firstly, from another perspective of the DIT-FFT: an N-point FFT requires a total of $n = log_2 N$ stages of transformation. According to the recursive principle, the calculations in i-th stage need 2^{n-1-i} data pairs $\{X_0(k), X_1(k)\}$ to perform the same merge operation. To clearly demonstrate this behavior, the point-value expression of the sub-polynomials

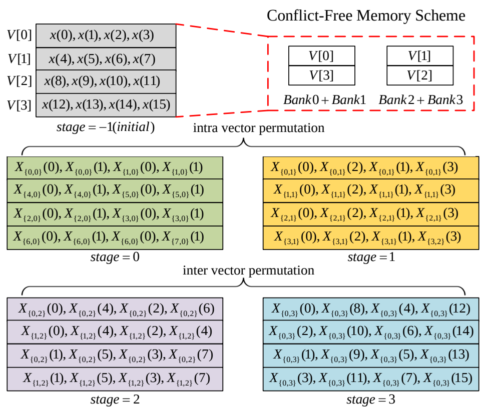

Conflict-Free Memory Scheme
V[0] x(0), x(1), x(2), x(3)
V[1] x(4), x(5), x(6), x(7)
V[2] x(8), x(9), x(10), x(11)
V[3] x(12), x(13), x(14), x(15)
V[0]
V[3]
V[1]
V[2]
Bank0 + Bank1
Bank2 + Bank3
stage = −1(initial)
intra vector permutation
X{0,0}(0), X{0,0}(1), X{1,0}(0), X{1,0}(1)
X{4,0}(0), X{4,0}(1), X{5,0}(0), X{5,0}(1)
X{2,0}(0), X{2,0}(1), X{3,0}(0), X{3,0}(1)
X{6,0}(0), X{6,0}(1), X{6,0}(0), X{7,0}(1)
stage = 0
X{0,1}(0), X{0,1}(2), X{0,1}(1), X{0,1}(3)
X{1,1}(0), X{1,1}(2), X{1,1}(1), X{1,1}(3)
X{2,1}(0), X{2,1}(2), X{2,1}(1), X{2,1}(3)
X{3,1}(0), X{3,1}(2), X{3,1}(1), X{3,2}(3)
stage = 1
inter vector permutation
X{0,2}(0), X{0,2}(4), X{0,2}(2), X{0,2}(6)
X{1,2}(0), X{1,2}(4), X{1,2}(2), X{1,2}(4)
X{0,2}(1), X{0,2}(5), X{0,2}(3), X{0,2}(7)
X{1,2}(1), X{1,2}(5), X{1,2}(3), X{1,2}(7)
stage = 2
X{0,3}(0), X{0,3}(8), X{0,3}(4), X{0,3}(12)
X{0,3}(2), X{0,3}(10), X{0,3}(6), X{0,3}(14)
X{0,3}(1), X{0,3}(9), X{0,3}(5), X{0,3}(13)
X{0,3}(3), X{0,3}(11), X{0,3}(7), X{0,3}(15)
stage = 3

Figure 7: 16-points FFT vector data flow with vector size $l = 4$. The result of each stage of the butterfly calculation in the vector transformation needs to be output permuted.

at point k is defined as follows: $X_{\{j,i\}}(k) = \sum_{r=0}^{\frac{N}{2^{n-i}}-1} x(j + r \cdot 2^{n-i}) W_N^{(j+r\cdot 2^{n-i})k}$, where i denotes the transformation stage, and j indicates the index of the polynomial in this stage.

$$
\begin{aligned}
X(k) &= \sum_{n=0}^{N-1} x(r) W_N^{kr}, \quad k = 0, 1, \ldots, N-1 \\
&= X_{\{0,n-1\}}(k) + X_{\{1,n-1\}}(k) \\
&= \left\{X_{\{0,n-2\}}(k) + X_{\{2,n-2\}}(k)\right\} + \left\{X_{\{1,n-2\}}(k) + X_{\{3,n-2\}}(k)\right\}
\end{aligned}
\tag{3}
$$

In Equation 3, $X_{\{j,i\}}$ represents the two sub-polynomials after splitting operation. It can be seen that in the i-th order transformation, the paired polynomials are $X_{\{j,i\}}$ and $X_{\{j+2^{n-1-i},i\}}$. And a permutation network for the coefficients is introduced. For the input vectors $\mathbf{a} = [a_0, \ldots, a_{l-1}]$ and $\mathbf{b} = [b_0, \ldots, b_{l-1}]$, the permutation results are $\tilde{\mathbf{a}} = [a_0, b_0, \ldots, a_{l/2-1}, b_{l/2-1}]$ and $\tilde{\mathbf{b}} = [a_{l/2}, b_{l/2}, \ldots, a_{l-1}, b_{l-1}]$. Following this strategy, the new FFT computation flow is obtained as shown in Figure 7. The phase $stage = -1$ represents the initial data's vector distribution. Stages 0 and 1 show the impact of intra-vector permutation on the distribution of the computational results (when $i < \log_2 l$, the output permutation operation occurs within the vector). Stages 2 and 3 depict the inter-vector output permutation because at this stage the coefficient distribution of the sub-polynomials $X_{\{j,i\}}$ extends beyond the range of a single vector. The result distribution after the intra-vector permutation and specifically the transformation resulting of the d-th $(d = log_2 l - 1)$ stage can be summarized as follows:

$$
\mathbf{v}[k] = X_{\{\frac{k}{\frac{N}{l \times log_2 l}} + [k mod \frac{N}{l \times log_2 l}] \times l, d\}}, k \in [0, 1, 2, \cdots, N/l - 1] \tag{4}
$$

Based on Figure 7 and the distribution of changes between vectors, the address intervals of the input vectors for butterfly computation are all powers of 2. Utilizing this property, the conflict-free logical-to-physical address mapping scheme is designed.

Then, this sub-polynomial can be stored in the memory continuously. For a dimension of m sub-polynomial (at the i-th stage transformation, $m = 2^{n-1-i}$), the order of coefficient storage is:

$$
X_{\{j,i\}}(k) = \left\{X_{\{j,i\}}(\texttt{BitRev}(0)), X_{\{j,i\}}(\texttt{BitRev}(1)), \cdots, X_{\{j,i\}}(\texttt{BitRev}(m-1))\right\} \tag{5}
$$

`BitRev` denotes the bit-reversal operation. This leads to the corresponding twiddle factor generation method, as depicted in Algorithm 3. The proposed FFT scheme reduces the storage consumption and optimizes the loading process of twiddle factors by combining the permutation network with the address generation method in Algorithm 3. During the first $d = \log_2 l$ stages, twiddle factors are read only once to be reused for each stage's remaining cycles. 74.6% memory accesses are avoided compared to the unreused method for 256-point FFT. This is similar to the "Early Shuffling" technique for NTT transformations on AVX architecture in [Sei22].

The whole vectorized FFT scheme proposed is as shown in the Algorithm 4. In the algorithm, `CircleRightShift`(x, y, z) denotes the circular right shift function. Specifically, it represents the bits obtained after circularly shifting z-bit x-variable y bits to the right.

Algorithm 3 Generation of Twiddle Factor for Vectorized FFT Scheme

```
Require: polynomial of degree $N$: $x$, vector dimension: $l$
Ensure: factor twiddle: $\boldsymbol{\omega}$
 1: $n = log_2 N,\ d = log_2 l$
 2: for $i = 0; i < d; i++$ do
 3:     $\boldsymbol{\omega}[i] = [W_{2^i}^{\texttt{BitRev}(0)}, \cdots, W_{2^i}^{\texttt{BitRev}(2^i-1)}]^{\times(l/2^i)}$    ▷ $\times\left(\frac{l}{2^i}\right)$ denotes that the vector is
    replicated $\frac{l}{2^i}$ times
 4: end for
 5: for $i = d; i < n; i++$ do
 6:     for $j = 0; j < 2^{i-d}; j++$ do
 7:         $\boldsymbol{\omega}[i+j] = [W_{2^i}^{\texttt{BitRev}(j\times l)}, \cdots, W_{2^i}^{\texttt{BitRev}((j+1)\times l-1)}]$
 8:     end for
 9: end for
```

Algorithm 4 Vectorized FFT Scheme

```
Require: Polynomial of degree $N$: $x$, twiddle factor: $\boldsymbol{\omega}$, vector dimension: $l$
Ensure: FFT$(x(k))$
 1: $n = log_2 N, d = log_2 l$
 2: for $i = 0; i < N/l; i++$ do
 3:     $\mathbf{v}[i] = x[i \times l : (i+1) \times l - 1]$
 4: end for
 5: $space = 2^{n-d-1}$, $m = 0$
 6: for $i = 0; i < n; i++$ do
 7:     $rotate\_num = i\%(n-d)$
 8:     $distance$ = CircleRightShift$(space, rotate\_num, n-d)$
 9:     for $j = 0; j < 2^{n-d-1}; j++$ do
10:         $addr$ = CircleRightShift(BitRev$(j)$, $rotate\_num, n-d)$
11:         $\mathbf{v}1 = \mathbf{v}[addr], \mathbf{v}2 = \mathbf{v}[addr + distance]$
12:         VectorBFU$(\mathbf{v}1, \mathbf{v}2, \boldsymbol{\omega}[m])$                ▷ Vector BFU operation
13:         for $k = 0; k < l/2; k++$ do                      ▷ Output Permutation
14:             $\mathbf{v}1\_wr[2k] = \mathbf{v}1[k]$
15:             $\mathbf{v}1\_wr[2k+1] = \mathbf{v}2[k]$
16:             $\mathbf{v}2\_wr[2k] = \mathbf{v}1[k + l/2]$
17:             $\mathbf{v}2\_wr[2k+1] = \mathbf{v}2[k + l/2]$
18:         end for
19:         $\mathbf{v}[addr] = \mathbf{v}1\_wr, \mathbf{v}[addr + distance] = \mathbf{v}2\_wr$
20:         $m = m + i >> (n - d - 1 - (i - d))$            ▷ Reusing the Twiddle Factor
21:     end for
22:     $m = m + 1$
23: end for
```

Comparision with related works. Compared to the previous vectorized FFT/NTT schemes, the proposed FFT architecture offers better scalability, fewer permutation networks, and efficient reusability for other arithmetic operations in Falcon at the minimal resource cost. The number of permutation networks are reduced from 6 to 2 compared to the work in [XHY+20] for 8-parallel FPU. A detailed correctness proof of the vectorized FFT architecture with any parallelism is firstly provided in this work, which has not been provided in [ZZL+22]. The other specific `splitfft`/`mergefft` operators in Falcon are also adapted to the vectorized FFT scheme with minimal overheads, which is detailed in Section 4.2.2.

Algorithm 5 Vectorized `mergefft` Scheme

```
Require: FFT result of the current polynomial $x(k)$ with degree $N$,degree of the initial
    FFT result: $n_0 = 256$, twiddle factor: $\boldsymbol{\omega}$, vector dimension: $l = 4$
Ensure: splitfft($FFT(x(k))$)
 1: twiddle_factor_rd = $[0, 1, 2, 3, 5, 9, 17, 33, 65, 129, 257, 513]$
 2: rotate_rd = $[0, 0, 0, 0, -1, -1, -1, 0, 1]$
 3: rotate_wr = $[0, 0, 0, 0, -1, -1, -1, 0, 1]$
 4: space_rd = $[0, 0, 0, 1, 1, 1, 1, 16, 16]$
 5: $n = log_2 N, d = log_2 l$
 6: $space$ = space_rd$[n]$
 7: $distance$ = CircleRightShift($space, rotate\_num, n - d$)
 8: for $i = 0; i < 2^{n-d-1}; j++$ do
 9:     $rd\_addr$ = CircleRightShift(BitRev($j$), rotate_rd$[i]$, $n - d$)
10:     $wr\_addr$ = CircleRightShift(BitRev($j$), rotate_wr$[i]$, $n - d - 1$)
11:     $\mathbf{v}1 = \mathbf{v}[rd\_addr], \mathbf{v}2 = \mathbf{v}[rd\_addr + distance]$
12:     VectorBFU($\mathbf{v}1, \mathbf{v}2, \boldsymbol{\omega}[m]$)
13:     for $k = 0; k < l/2; k++$ do                         ▷ Output Permutation
14:         $\mathbf{v}1\_wr[2k] = \mathbf{v}1[k]$
15:         $\mathbf{v}1\_wr[2k+1] = \mathbf{v}2[k]$
16:         $\mathbf{v}2\_wr[2k] = \mathbf{v}1[k + l/2]$
17:         $\mathbf{v}2\_wr[2k+1] = \mathbf{v}2[k + l/2]$
18:     end for
19:     $\mathbf{v}[wr\_addr] = \mathbf{v}1\_wr, \mathbf{v}[wr\_addr] = \mathbf{v}2\_wr$
20:     $m = m$ + twiddle_factor_rd$[n] + i$
21: end for
```

4.2.2 Hardware-reusage of Splitfft and Mergefft Processing

Naturally, the individual `splitfft` and `mergefft` are adapted to the FFT/IFFT architecture and vectorized scheme. As analyzed in Section 4.2, `mergefft` and `splitfft` are sub-steps of FFT and IFFT, respectively. In our indexing scheme Section 4.2.1, this can be represented as follows ($X_{\{0,i\}}$ as an example in an N-points FFT with vector dimension of l):

$$\begin{aligned}\texttt{splitfft}(X_{\{0,i\}}) &= X_{\{0,i-1\}} + X_{\{0+2^{n-i},i-1\}} \\ \texttt{mergefft}(X_{\{0,i-1\}}, X_{\{0+2^{n-i},i-1\}}) &= X_{\{0,i\}}\end{aligned} \tag{6}$$

where i denotes the i-th stage transformation. Therefore, the merging operation of any two paired sub-polynomials is `mergefft`. Correspondingly, splitting any polynomial into two sub-polynomials becomes `splitfft`. Thus, in practical implementation, it is only needed to implement the splitting and merging of sub-polynomials $X_{\{0,i\}}$. Consequently, during adaptation, it only needs to design an address generation pattern for $X_{\{0,i\}}$. The specific implementation can be seen in the Algorithm 5.

The vectorized scheme of `mergefft` differs from the vector scheme of FFT because it eliminates the reuse of twiddle factors. To implement this function in the FFT/IFFT architecture with minimal cost, the input addresses are pre-computed for $X_{\{0,i\}}$ at each stage of `splitfft`/`mergefft`. The generation of addresses involves **rotate_rd**, **rotate_wr**, and **space_rd** in the Algorithm 5. For the twiddle factor processing, it is needed to mark the initial read address of the i-th stage, which is finally integrated into **twiddle_factor_rd**.

Finally, memory conflicts are resolved by using cross-storage. These read-write conflicts occur when $X_{\{0,i\}}$ stores the two sub-polynomials output by the `splitfft` operation in the normal arrangement, resulting in both vectors written to the same banks simultaneously. The interleaving storage method is adopted to solve this problem. Specifically, the address generation is mirrored for $X_{\{1,2\}}$, so the data that should be written to Bank0+Bank1 is written to Bank2+Bank3, and vice versa, forming a interleaving-storage with $X_{\{0,2\}}(0)$ to avoid the conflict. The original storage is defined as the "Positive Set", while the interleaving storage is referred to as the "Negative Set". This interleaving storage implementation only requires computing 1-bit signal `bank_sel` to change the bank selection.

4.2.3 Pipelined BFU and Parallelism Evaluation

As mentioned in 4.2, it is natural to use a unified butterfly unit architecture supporting multiple functions, rather than multiple modules that each support a single function. The designed unified butterfly unit is shown in Figure 6. For vectorized operations involving complex number arithmetic, the FPU module supports batch calculations with different vector widths of single-bank, dual-bank, and four-bank. The interleaving-storage addresses of input operands proposed in 4.2.2 are adopted in the four-bank vectorized calculations.

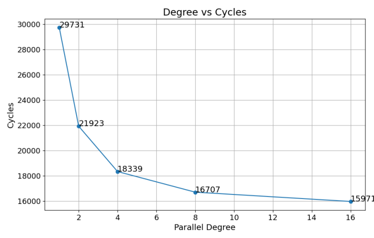
Floating-Point Unit Optimization. Double-precision floating-point operations [Kah96] are involved in Falcon, but the special values (Infs, NaNs, and subnormals) are not needed to be handled due to the Falcon's algorithm characteristic [PFH+20]. This allows for further optimizations of fpr_mul, fpr_add, and fpr_sub hardware modules. The design without special values handling reduces the area overheads much and saves DSP resources in FPGA implementations. After optimization, the DSPs consumption of a single FPU is decreased by 47%. Additionally, the utilization of DSPs can also be optimized for constant multiplication. Optimization results are shown in Table 2.

Table 2: Optimization results and comparisons of floating-point modules.

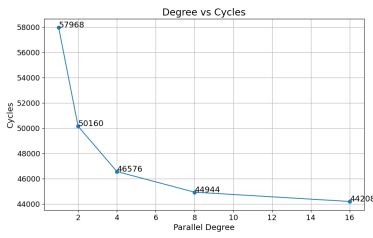
	ASIC (μm^2 / MHz)		FPGA (LUTs/FFs/DSPs / MHz)	
	DesignWare 2019 IP	FalconSign Opt, **this work**	Vivado 2022.2 IP	FalconSign Opt, **this work**
fpr add	2158 / 500	1992 / 500	882/260/ 0 / 247	600/193/0 / 235
fpr mul	9843 / 500	8173 / 500	122/102/13 / 212	219/129/9 / 275

Parallelism Evaluation. The parallelism of BFUs is determined by assessing the performance improvements of different BFU numbers for the same computation workload. The calculations of FPU are involved in the data preprocessing/postprocessing, `ffSampling` and `ffLDL`. Based on the proposed FFT hardware, the cycle amount of FPU-related calculations are estimated accurately for different parallelisms of BFUs. Figure 8 shows that the design of 4-parallel BFUs lies in the turning point of the performance curve and improves the ASIC performance by 38%, while higher-parallel BFUs yield less than 5% additional gains. Therefore, the parallelism of 4 is adopted in our design to achieve high-throughput Falcon signature generation with reasonable overheads.

Comparisons with related works. The proposed floating-point modules features a compact pipeline design and higher frequency, with optimizations for Falcon's floating-point computation characteristics. Compared to the open-source Berkeley floating-point module



(a) Falcon-512-ASIC-FPU



(b) Falcon-512-FPGA-FPU

Figure 8: The performance curve of the FPU-related calculations in Falcon-512 with the parallelism degree of BFUs.

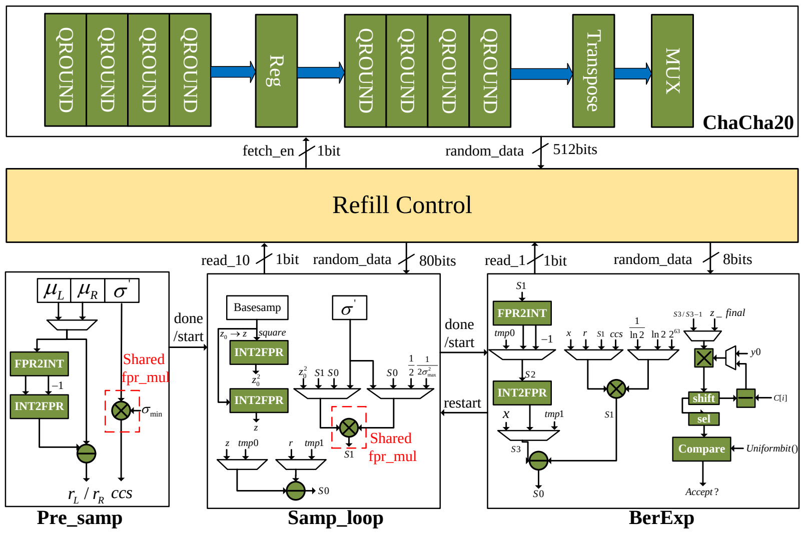


Figure 9: Dedicated SamplerZ Module.

used in [YSZ$^+$24], the handling logic for special values is saved and the maximum frequency increases from 83 MHz to 185 MHz on the same device. Compared to [LYN$^+$24], the frequency increases from 300 MHz to 500 MHz under the same 28nm process benefiting from the optimized design. Additionally, the pipeline depth of multiplication hardware is reduced from 5 stages to 2 stages, which decreases the computation latency.

4.3 Low-Latency Gaussian Sampler

The Gaussian sampler performs the discrete Gaussian sampling operation, with the output as an integer $z \sim D_{\mathbb{Z},\sigma',\mu}$. Generating a valid signature at security level I/V at least requires 1024/2048 samples. The input μ for each sampling is derived from the message m, the private key sk, and the result of the previous signature, while σ' is part of the private key. It can be seen that the calculations of these samples are dependent on the previous samples, which forms a computational bottleneck. Hence, the sampler hardware is optimized for low latency to reduce the whole execution cycles.

The proposed Gaussian Sampler is shown in Figure 9. The Sampler consists of the ChaCha20, Refill_Control, Pre_samp, Samp_loop, and BerExp modules. ChaCha20 and Refill_Control are responsible for providing random numbers in real-time, while the remaining modules execute the sampling computation. The key computational flow for low-latency sampling is designed with balanced pipelining. Therefore, the sampling operation is divided into three stages: Pre_samp module computes $r \leftarrow \mu - \lfloor \mu \rfloor$ and $ccs \leftarrow \sigma_{\min}/\sigma'$; Samp_loop module computes `BaseSampler()` function and $x \leftarrow \frac{(z-r)^2}{2\sigma'^2} - \frac{z_0^2}{2\sigma_{\max}^2}$; BerExp module executes the `BerExp()` function. The three computational processes correspond to the three modules in `SamplerZ`.

For the Pre_samp part, $\lfloor \mu \rfloor$ and $\sigma_{\min}/\sigma'$ are not hardware-friendly. For $\lfloor . \rfloor$, it cannot be directly implemented. Generally, when converting a floating-point number to a fixed-point number, the `Rounding` rule is obeyed to convert the floating-point number to the nearest integer, which is not necessarily the floor value. Therefore, a prediction mechanism is proposed to resolve this problem in combination with the computation $r \leftarrow \mu - \lfloor \mu \rfloor$. $r_0 \leftarrow \mu - \texttt{Rounding}(\mu)$ and $r_1 \leftarrow \mu - (\texttt{Rounding}(\mu) - 1)$ are computed consecutively to judge the sign bit of r_0. If it is zero, $r = r_0$; otherwise, $r = r_1$. The second optimization occurs in the Sign_Tree singing method, where $1/\sigma'$ is computed during the pre-computation of the expanded form of the private key. This approach effectively converts floating-point division into multiplication, aligning with the Falcon's documentation [PFH+20]. For the floating-point multiplication unit, even if the sampling is rejected, Pre_samp does not need to recompute. Samp_loop and BerExp need to recompute. The calculations between these two parts are not overlapped. Therefore, Pre_samp and Samp_loop modules can share the multiplication unit.

Besides, the Samp_loop part is optimized to reduce the latency. The main method is to parallelize subtraction, multiplication, and `BaseSampler()`. First, $\frac{1}{\sigma'^2}$ is hidden by the `BaseSampler()` operation time. Second, $(z - r)$ is computed in parallel with $\frac{1}{2\sigma'^2}$.

The BerExp module adopts a dedicated multiplication unit and calculates the required floating-point calculations through time-division multiplexing. Additionally, for the $\lfloor x/\ln 2 \rfloor$ operations, the same prediction mechanism as Pre_samp is used. For $z \leftarrow \lfloor 2^{63} \cdot x \rfloor$, it is immediately followed by the operation $y \leftarrow \mathcal{C}[u] - (z \cdot y) \gg 63$. To reduce the latency, the same operation is performed with $z \leftarrow \texttt{Rounding}(2^{63} \cdot x) - 1$ as input. Finally, the prediction mechanism determines the correctness of the output.

The whole sampling process has a cycle amount of 67-74. The computation delays for Pre_samp, Samp_loop, and BerExp are 11 cycles, 16 cycles, and 40-47 cycles, respectively. The fluctuation in BerExp's delay is caused by rejections. The peak usage of random numbers occurs from the start of Samp_loop to BerExp rejection, followed by immediately re-entering Samp_loop for resampling until BerExp uses random numbers again. In this process, 224 random bits are consumed in 103 cycles (16+47+40). As shown in the Figure 9, `ChaCha20` is used to generate random numbers, and Refill_Control manages the random number supply. Refill_Control has a 256-bit buffer to prefetch random numbers, ensuring real-time supply. The buffer adopts the FIFO strategy to prefetch random bits. When new random numbers are needed, `ChaCha20` can generate them within 80 cycles, which can meet the peak usage demand. ChaCha20 module requires 10 rounds of transformation to generate a set of random numbers. Each round needs eight- `QROUND` operations. For consistency with the official software, eight sets of random numbers are generated at once and then are transposed before output. In the application scenario of the FalconSign accelerator, ChaCha20 module needs a true-random number generator to generate seeds for initializing ChaCha20.

Latency Optimization and Comparisons. Two optimization techniques for low-latency sampling are proposed: 1. Merge the consecutive samples with the same σ' input to hide the Pre_samp phase of the second sample and the sampling cycles is reduced up to 7%; 2. Overlap the computations in Pre_samp and Samp_loop, enabling the part of the Samp_loop calculations to start three cycles in advance. Based on these optimizations, the design implements pipelined optimization and time hiding for certain sampling phases, reducing the latency of single sampling to 74 cycles and double sampling to 137 cycles unless rejected. This is significantly lower than existing samplers, which require 1807 cycles [KA24] and 104 cycles [YSZ+24] for single sampling. Additionally, on the Artix-7 and Zynq UntraScale platform, perfermance improvements of 13.1× and 3.1× were achieved compared to [KA24] and [YSZ+24], respectively. To the best of our knowledge, this represents the lowest sampling latency (74 cycles per sample), achieving a maximum frequency of 185 MHz on FPGA.

4.4 Other Modules

The remaining modules include FPR2INT, INT2FPR, and Check_Sig modules. INT2FPR is mainly used to convert the polynomial c generated by the HashToPoint rejection sampling into double-precision floating-point format. Conversely, FPR2INT converts the valid signature polynomial s_2 from floating-point format to 16-bit fixed-point format and then sends it to the Encoder module for encoding. These two modules only need to be executed once in each signature generation process. The floating-point to fixed-point conversion circuits used are the IP provided by Xilinx Vivado, with a conversion parallelism of 4.

The Check_Sig module is used to handle the boundary check of the signature. As described above, the generated signature must be small enough to be considered valid. This criterion is $\|\mathbf{s}\|^2 \leq \lfloor \beta^2 \rfloor$. The computation of $\|\mathbf{s}\|^2$ is essentially the sum of the squares of each element of s_2, s_1, denoted by the summation symbol $\sum s_{2_i}^2 + s_{1_i}^2$.

5 Implementation and comparison

The proposed architecture is described in System Verilog and is simulated, synthesized, and implemented using Xilinx Vivado for the target platform Xilinx ZCU104 board that has an XCZU7EV-2FFVC1156 FPGA. Additionally, our architecture is evaluated on the TSMC 28nm HPC platform.

5.1 Resources Usage and Performance Results

The resource consumption on FPGA of the whole design and major modules are shown in Table 3. The utilization ratios of LUTs/FFs/DSPs/BRAMs for Falcon-512/1024 on the FPGA platform is 34%/11%/13%/14% and 34%/11%/13%/18%, respectively. The FPU core accounts for more than 60% LUTs, 55% FFs, and 65% DSPs of the total consumption. This is because the high-speed reconfigurable BFU array in the FPU consumes 16/16/32 expensive double-precision floating-point Add/Sub/Mul units to match the data path bandwidth. The second most area-consuming module is the SamplerZ core, which occupies 18% LUTs, 22% FFs, and 76 DSPs of the total consumption. This is for balancing the critical path of the complex computation pipeline and hiding some computations, resulting in significant overheads but greatly reducing the sampling latency. Additionally, the ChaCha20 random number generator in the Sampler consumes 4096-bit registers to transpose random numbers, ensuring consistency with the official software implementation. The storage consumption is mainly occupied by the Data memory module, which stores the large private key and temporarily stores the intermediate values in the tree-like computations. The storage usage employs dual-port memory to reduce waiting times during FPU computations. Compared to the utilization of single-port memory on the ASIC platform, this approach saves 5% cycle number.

The performance and resource consumption results of the implementation are shown in Table 3. The accelerator can operate at 185MHz after placing and routing. Table 3 lists the computation cycles of the main modules, with each module corresponding to a computation phase of the protocol, as illustrated in Figure 2. The FPU and SamplerZ consume 30% and 65% of the computation time, respectively. After deep optimization,

Table 3: Resource consumption and cycle consumption of the major module in Falcon-512/Falcon-1024.(Frequency: 185MHz)

Module	**LUTs**	**FFs**	**DSPs**	**BRAMs**	**Cycles**
FPU	45859/44263	24901/24920	144/144	0/ 0	47090/ 95202
SamplerZ	14710/14723	10731/10550	76/ 76	0/ 0	104947/206268
SHAKE256	7286/ 7572	3756/ 3761	0/ 0	0/ 0	201/ 381
HashToPoint	1623/ 1630	263/ 259	0/ 0	0/ 0	143/ 280
INT2FPR	694/ 714	687/ 687	0/ 0	0/ 0	140/ 268
FPR2INT	844/ 861	1076/ 1075	0/ 0	0/ 0	280/ 536
Data Memory	1286/ 1296	2158/ 2168	0/ 0	45/58	-
Total	80497/80496	46768/46495	220/220	45/58	160060/317450
Utilization Ratio	34%/34%	11%/11%	13%/13%	14%/18%	—

the FPU still occupies a significant proportion of the computation time, reflecting that floating-point vector calculations in `ffSampling` are the primary computational bottleneck. For the SamplerZ module, discrete Gaussian sampling remains the largest computational bottleneck in high-speed scenarios, consuming over half of the computation time.

Our design is also evaluated on the TSMC 28nm process through Design Compiler-2022.03 for the evaluation of extremely low-latency and high-speed scenarios. On the ASIC platform, the proposed architecture of Falcon-I occupies an area of $0.71\,\mathrm{mm}^2$ (759.0K gates + 136.3KB SRAM) and operates at a frequency of 530MHz, while the accelerator of Falcon-V occupies $0.97\,\mathrm{mm}^2$ (771.6K gates + 272.6KB SRAM) with 500MHz. The average power is 62.0 mW/68.7 mW and the energy consumption is 11.8uJ/26.1uJ for Falcon-512/Falcon-1024, respectively. The signature generation of our work consumes 99k/188k cycles, achieving a latency of $0.19ms/0.38ms$ and throughput of 5.2k/2.6k OPS for Falcon-512/Falcon-1024, which can adapt to most of the scenarios of high-speed signature generation. The reduction in cycle counts benefits from an optimized pipelined design for the primary bottleneck modules, FPU, and `SamplerZ`.

Sign_Tree vs. Sign_Dynamic. Sign_Tree mode generates the signature by pre-expanding the private key into a "Falcon tree", while Sign_Dynamic mode dynamically expands the private key to save storage consumption. The resource usage for Sign_Dynamic for Falcon-512/1024 on the FPGA platform is 99949/53578/220/45 and 100442/49771/220/55, respectively, with an implementation frequency of 158 MHz and signature cycles of 264k/527k. For Sign_Dynamic, the support for dynamic computation of the "Falcon tree" introduces floating-point division and square root modules, resulting in an increase in LUTs/FFs/DSPs/BRAMs and an cycle account increase of 65%. In the software platform, Sign_Dynamic also causes a performance drop of 103% [Por19]. The utilization of BRAMs is reduced by 10% in Falcon-1024 as the memory can be reused to dynamically generate the "Falcon tree."

5.2 Comparisons with Related Works

In Table 4, our architecture is compared to the most recent implementations of post-quantum signature algorithms. Due to differences in implemented algorithms, security levels, implementation platforms, and design methodologies, a fair comparison is not always possible. Nonetheless, our architecture achieves a remarkable acceleration effect for Falcon signature generation while consuming a moderate amount of area and resources. To the best of our knowledge, this architecture is also the first fine-grained optimized full hardware implementation for Falcon signatures and it is the fastest signature generation hardware architecture with the best ATP.

Table 4: Comparison results of high-speed Signature generation of Falcon. (SW: software implementation; HW: full hardware implementation; HLS: high-level synthesis; HW/SW denotes hardware-software co-design.)

	Level	Area (LUT/FF/DSP/BRAM) (or mm^2 for ASIC)	Freq (MHz)	Cycles	ATP (LUT/FF/DSP/BRAM) (or mm^2 for ASIC)
SW [NG23] M1 (ARMv8)	I	—	3200	442k	—
	V			882k	
SW [PFH+20] Core i5 (AVX2)	I	—	2300	389k	—
	V			790k	
HW/SW [KA24] (Artix-7)	I [c]	(3683/2107/-/2) [a]	121	50407k	$22.2\times/21.8\times/-/21.5\times$
HW/SW [YSZ+24] (Zynq UltraScale+)	I	(18565/3210/17/0) [a]	83	1554k	$5.1\times/1.5\times/1.7\times/-$
	V			3188k	$5.1\times/1.5\times/1.7\times/-$
HW/SW [LYN+24] (Samsung 28nm)	I	0.038 [a]	300	6591k	10.8×
	V			14078k	8.4×
HW (HLS) [SAW+23] (Zynq UltraScale+)	I	46971/44249/182/32	187.5	788k	$2.8\times/4.6\times/4.0\times/3.5\times$
	V	45223/41370/182/37		1638k	$2.8\times/4.5\times/4.2\times/3.2\times$
HW **This work** (Artix-7)	I	**79065/46389/226/45**	**65**	**160k**	—
HW **This work** (Zynq UltraScale+)	I	**80497/46768/220/45**	**185**	**160k**	$1.0\times/1.0\times/1.0\times/1.0\times$
	V	**80496/46495/220/58**		**320k**	
HW **This work** (TSMC 28nm)	I	**0.71(0.39(L)+0.32(M))** [b]	**530**	**99k**	$1.0\times/1.0\times/1.0\times/1.0\times$
	V	**0.97(0.40(L)+0.57(M))** [b]	**500**	**188k**	

[a] For the HW/SW implementation, the resource consumption and ATP are calculated only based on the hardware part, without considering the area consumption of general processors.
[b] L: logic, M: Memory.
[c] The security level is not clearly stated in their paper, Level-I is inferred.

First, compared to the software implementation, FalconSign achieves comparable performance to [NG23] and [PFH+20] with a smaller area. FalconSign requires 0.19 ms to complete the signature generation for Falcon-512, while the Apple M1 (ARMv8) platform [NG23] requires 0.14 ms, and the Intel® Core® i5-8259U (Coffee Lake) platform [PFH+20], using AVX2 and FMA, requires 0.17 ms. It is to be noted that FalconSign works at a frequency of 500 MHz, significantly lower than 3.2 GHz and 2.3 GHz, and the execution cycles are reduced by 4.46× and 3.92×, respectively.

Second, the existing HW/SW implementations of Falcon signature generation are compared to our work. The work by Karabulut [KA24] accelerates the discrete Gaussian sampling using an HW/SW co-design approach. Compared to this work, our signing performance is improved by 484.4×. Our design reduces the sampling cycle count by 96.0× and the signing cycle count by 315.0×. Furthermore, to ensure a fair comparison, Falcon-512 is also implemented on the Artix-7, consuming LUTs/FFs/DSPs/BRAM/s of 79065/46389/226/45, with a frequency of 65 MHz. This achieves an improvement over the ATP in [KA24] by 7.9×/7.7×/-/7.5×. And the signing speed has increased by 169 ×. The performance improvement benefits from our low-latency Gaussian sampler, which achieves a lower sampling delay. [KA24] focuses on the large Subtraction-Multiplication operation in `BerExp`: $y = c - (z \times y) \gg 63$. However, the delay of large-digit multiplication in their design is 8 cycles, with a throughput of one multiplication every 4 cycles, which is not suitable for high-speed scenarios. In our work, the same expected value μ for two consecutive samplings is utilized, thereby the r and ccs calculations of one sampling are optimized, leading to a better overall sampling improvement. Moreover, in the entire signing process, the bottleneck of floating-point vector calculations is addressed. The `ffSampling` operation accounts for 90% of the signing time, with FPU and sampling being the two main operations. Our vectorized FPU architecture and algorithm reduce its computation time in `ffSampling` from 75% in [KA24] to 31%.

Yu et al. [YSZ$^+$24] reported a HW/SW implementation based on the RSIC-V extended instruction set. The work proposed a RISC-V scalar-vector framework and a discrete Gaussian sampling circuit for efficient Falcon implementation. Although it considered the sampling bottleneck, it does not accelerate floating-point calculations in `ffSampling`. Furthermore, the pipeline design of their sampler is not balanced, achieving a delay of 104 cycles for single sampling at a frequency of only 83MHz. In contrast, our sampling delay is just 74 cycles at a frequency of 185MHz. Additionally, their work utilized the floating-point module to calculate integer operations in `BerExp`, which increases the complexity of the implementation. Compared to this work, our design improves the signing performance of Falcon-512/Falcon-1024 by 21.8×/22.3×, respectively. Even if only the resource consumption of their sampling circuit is considered, our design's ATP is improved by 5.1×/1.5×/1.7× for both I/V security levels. In terms of key performance results, a higher frequency of 185MHz is achieved, and the computation cycles for signing are reduced by 9.7×/10.0×.

Lee et al. [LYN$^+$24] reported a HW/SW co-design architecture based on Cortex M4 and ASIC. Compared to this work, our architecture achieves an acceleration of 202.5×/215.1× in throughput and ATP improved by 10.8×/8.4× on the 28nm process. It is to be noted that the comparisons only include the hardware part of [LYN$^+$24]. Considering the energy efficiency, our architecture improves 700.9×/677.4× when both the software (229mW) and hardware (5.8mW) parts [LYN$^+$24] are included. The results indicate that our design is also suitable for the energy-saving scenarios.

Schmid et al. [SAW$^+$23] utilized the HLS method to implement Falcon-`Sign`. Although this implementation provides some preliminary evaluations of Falcon hardware, it lacks in-depth architectural explanation and exhibits some unknown issues. For instance, when instantiating the `ChaCha20` module, three modules were instantiated, but only one was used. Compared to this work, our implementation achieves a 4.9×/5.1× performance improvement for the I/V security levels, with ATP improvements for LUTs/FFs/DSPs/BRAMs of 2.8×/4.6×/4.0×/3.5× and 2.8×/4.5×/4.2×/3.2×, respectively.

6 Conclusion

To the best of our knowledge, this paper proposed the first fine-grained full-hardware implementation for high-throughput/low-latency Falcon's signature generation. Memory-centric overall architecture, optimized tree-like Fast Fourier sampling flow, and several optimized modules are adopted in our design to fully adapt to Falcon's characteristics, which achieves the lowest execution latency and the best ATP compared to state-of-the-art works. Compared to previous software-hardware co-design and hardware solutions, our full-hardware implementation significantly reduces the execution latency and explores the boundaries of Falcon hardware implementation, which may promote its application in the fields of vehicle networking, payment settlement systems, etc. In the future, we will continue to support Falcon's key generation and consider the side-channel resistant design.

Acknowledgments

This work is supported in part by the National Key R&D Program of China (Grant No. 2023YFB4403500), and in part by the National Natural Science Foundation of China (Grant No. 62274102), and in part by the National Key R&D Program of China (Grant No.2021YFB2701201), and in part by the Postdoctoral Fellowship Program of the CPSF (Grant No.BX20240204). The authors thank the editors and reviewers for their thoughtful comments.

References

[BDF$^+$11] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. Random oracles in a quantum world. In *Advances in Cryptology–ASIACRYPT 2011: 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings 17*, pages 41–69. Springer, 2011.

[BDK$^+$18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[BHK$^+$19] Daniel J Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The sphincs+ signature framework. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 2129–2146, 2019.

[DP16] Léo Ducas and Thomas Prest. Fast fourier orthogonalization. In *Proceedings of the ACM on international symposium on symbolic and algebraic computation*, pages 191–198, 2016.

[Gar22] Mario Garrido. A survey on pipelined fft hardware architectures. *Journal of Signal Processing Systems*, 94(11):1345–1364, 2022.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 197–206, 2008.

[Gro96] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. Ntru: A ring-based public key cryptosystem. In *International algorithmic number theory symposium*, pages 267–288. Springer, 1998.

[KA24] Emre Karabulut and Aydin Aysu. A hardware-software co-design for the discrete gaussian sampling of falcon digital signature. In *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 90–100. IEEE, 2024.

[Kah96] William Kahan. Ieee standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996.

[LDK$^+$20] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. Crystals-dilithium. *Algorithm Specifications and Supporting Documentation*, 2020.

[LYN$^+$24] Yongseok Lee, Jonghee Youn, Kevin Nam, Heon Hui Jung, Myunghyun Cho, Jimyung Na, Jong-Yeon Park, Seungsu Jeon, Bo Gyeong Kang, Hyunyoung Oh, et al. An efficient hardware/software co-design for falcon on low-end embedded systems. *IEEE Access*, 2024.

[LZS$^+$24] Wai-Kong Lee, Raymond K Zhao, Ron Steinfeld, Amin Sakzad, and Seong Oun Hwang. High throughput lattice-based signatures on gpus: Comparing falcon and mitaka. *IEEE Transactions on Parallel and Distributed Systems*, 35(4):675–692, 2024.

[NG23] Duc Tri Nguyen and Kris Gaj. Fast falcon signature generation and verification using armv8 neon instructions. In *International Conference on Cryptology in Africa*, pages 417–441. Springer, 2023.

[oST15] National Institute of Standards and Technology. Post-quantum cryptography, 2015. Created January 03, 2017, Updated June 24, 2020.

[PFH$^+$20] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[PJSO24] Siddhartha Patra, Saeed S Jahromi, Sukhbinder Singh, and Román Orús. Efficient tensor network simulation of ibm's largest quantum processors. *Physical Review Research*, 6(1):013326, 2024.

[Por19] Thomas Pornin. New efficient, constant-time implementations of falcon. *Cryptology ePrint Archive*, 2019.

[RB20] Sujoy Sinha Roy and Andrea Basso. High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 443–466, 2020.

[SAW$^+$23] Michael Schmid, Dorian Amiet, Jan Wendler, Paul Zbinden, and Tao Wei. Falcon takes off-a hardware implementation of the falcon signature scheme. *Cryptology ePrint Archive*, 2023.

[SD3] NIST Sha and NIST DRAFT. standard: Permutation-based hash and extendable-output functions. *FIPS PUB*, 202:2015, 3.

[Sei22] Gregor Seiler. *Practical Lattice-Based Zero-Knowledge Proof Systems*. PhD thesis, ETH Zurich, 2022.

[Sho94] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[SKD20] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. Post-quantum authentication in tls 1.3: a performance study. *Cryptology ePrint Archive*, 2020.

[TBRM22] Geoff Twardokus, Nina Bindel, Hanif Rahbari, and Sarah McCarthy. When cryptography needs a hand: Practical post-quantum authentication for V2V communications. Cryptology ePrint Archive, Paper 2022/483, 2022. https://eprint.iacr.org/2022/483.

[XHY$^+$20] Guozhu Xin, Jun Han, Tianyu Yin, Yuchao Zhou, Jianwei Yang, Xu Cheng, and Xiaoyang Zeng. Vpqc: A domain-specific vector processor for post-quantum cryptography based on risc-v architecture. *IEEE transactions on circuits and systems I: regular papers*, 67(8):2672–2684, 2020.

[YSZ$^+$24] Xinglong Yu, Yi Sun, Yifan Zhao, Honglin Kuang, and Jun Han. Rvce-fal: A risc-v scalar-vector custom extension for faster falcon digital signature. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024.

[ZZL$^+$22] Yihong Zhu, Wenping Zhu, Chongyang Li, Min Zhu, Chenchen Deng, Chen Chen, Shuying Yin, Shouyi Yin, Shaojun Wei, and Leibo Liu. Repqc: A 3.4-uj/op 48-kops post-quantum crypto-processor for multiple-mathematical problems. *IEEE Journal of Solid-State Circuits*, 58(1):124–140, 2022.

[ZZO$^+$24] Yihong Zhu, Wenping Zhu, Yi Ouyang, Junwen Sun, Min Zhu, Qi Zhao, Jinjiang Yang, Chen Chen, Qichao Tao, Guang Yang, et al. 16.2 a 28nm 69.4 kops 4.4 μj/op versatile post-quantum crypto-processor across multiple mathematical problems. In *2024 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 67, pages 298–300. IEEE, 2024.

[ZZW$^+$22] Cankun Zhao, Neng Zhang, Hanning Wang, Bohan Yang, Wenping Zhu, Zhengdong Li, Min Zhu, Shouyi Yin, Shaojun Wei, and Leibo Liu. A compact and high-performance hardware architecture for crystals-dilithium. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 270–295, 2022.