Hardware Implementation of Discrete Gaussian Sampling for FALCON Digital Signature

Nhat Dang*, Padmanabha Nikhil Bhimavarapu*
*Department of Electrical and Computer Engineering, North Carolina State University
{nmdang,pbhimav}@ncsu.edu

***Abstract*—The advent of quantum computing threatens the foundations of classical cryptography, necessitating quantum-resistant alternatives. FALCON, a lattice-based digital signature scheme standardized by NIST, offers compact and secure solutions for post-quantum cryptography. A critical component of FALCON's signature generation process is Discrete Gaussian Sampling (DGS), accounting for a significant portion of computational overhead. This report presents a hardware implementation of DGS designed to enhance FALCON's performance, targeting the Xilinx Zynq SoC FPGA architecture. The proposed design employs a full hardware approach, partitioning base sampling and rejection sampling between programmable logic and processing systems. Experimental results demonstrate a speedup of approximately 29.5× in throughput compared to the software implementation. This scalable and efficient implementation showcases the potential of hardware acceleration in quantum-safe cryptographic operations for real-time and resource-constrained environments.**

***Index Terms*—Post-quantum cryptography, FALCON**

I. Introduction

Post-quantum cryptography [1], [2] has become increasingly crucial as quantum computing advances threaten traditional cryptographic systems. Among the quantum-resistant solutions, the FALCON [3] (Fast Fourier Lattice-based Compact Signatures over NTRU) digital signature scheme stands out as one of the few algorithms selected by NIST for standardization. A critical component of FALCON is its Discrete Gaussian Sampling (DGS) process, which accounts for up to 72% of the signature generation time. This project presents a hardware implementation of the DGS component for FALCON, targeting the Xilinx Zynq-7000 SoC FPGA platform. Our implementation focuses on optimizing the two-layer sampling mechanism: base sampling with precomputed values and rejection sampling for distribution adjustment. The implementation employs several optimization techniques, including a ChaCha20-based PRNG core, efficient memory management for the cumulative distribution table, and a specialized arithmetic pipeline for rejection sampling calculations. This fully hardware-based approach demonstrates the feasibility of implementing post-quantum cryptographic operations in embedded systems while maintaining the security guarantees required for quantum-resistant digital signatures.

II. Background and Prior Work

A. Lattice-Based Cryptography and PQC

Lattice-based cryptography (LBC) is a leading approach in post-quantum cryptography (PQC), valued for its balance of efficiency and adaptability. Among the four NIST-selected algorithms for PQC standardization, three—CRYSTALS-Kyber, CRYSTALS-Dilithium [4], and FALCON—are based on LBC. These schemes primarily rely on two core problems: the learning with errors (LWE) problem and the short integer solution (SIS) problem. The LWE problem involves reconstructing a vector s from a noisy equation $b = As + e$, where e represents an error sampled from a specific distribution. The SIS problem, in contrast, requires finding a nonzero vector $z \in \mathbb{R}^n$ such that $Az = 0$. FALCON leverages the SIS problem over NTRU lattices, which are structured using a polynomial pair $(f, g) \in R$, where $h = f/g \mod q$ defines the lattice. This framework enables FALCON to achieve both strong security and compact signature sizes.

B. FALCON

FALCON is a digital signature algorithm (DSA) based on the Gentry-Peikert-Vaikuntanathan (GPV) framework for constructing hash-and-sign lattice-based cryptography (LBC). It utilizes NTRU lattices and incorporates a trapdoor sampler that combines the efficiency of Peikert's algorithm with the quality of Klein's algorithm. Among the NIST PQC algorithms, FALCON is notable for achieving the smallest combined size of public and signature keys [5]. FALCON operates through three main procedures: key generation, signature generation, and signature verification. Since signature generation and verification are performed more frequently in scenarios where keys are reused, accelerating these operations yields significant benefits. While signature verification has been the focus of many acceleration studies, signature generation remains less explored. This work focuses on optimizing the signature generation process, which involves generating a hashed value c from a message m and a random r using the `HashToPoint` function. This is followed by repeated execution of the `ffSampling` function, which leverages FFT routines, and finally encoding the signature s through compression. This project aims to enhance the efficiency of these key steps in FALCON's signature generation procedure.

C. Fast Fourier Sampling (ffSampling)

Fast Fourier sampling (ffSampling) is a crucial component in the FALCON signature generation process. This method enhances performance by dividing polynomials using the Fast Fourier Transform (FFT). The ffSampling procedure involves

Algorithm 1 FALCON Signature Generation Algorithm

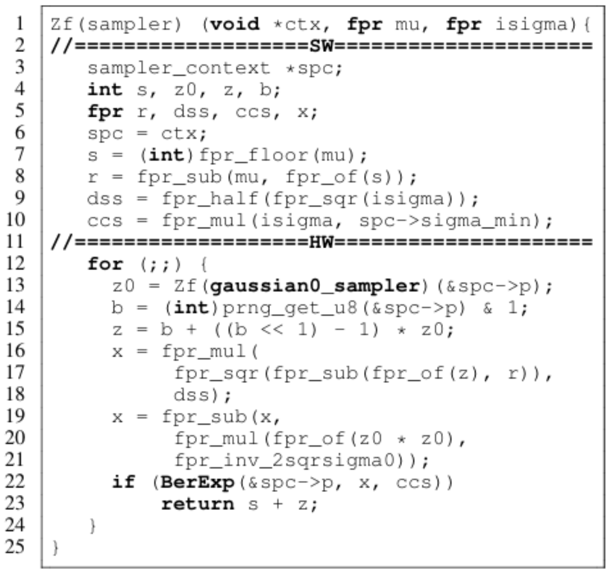
```
Input: message $m$, secret key $sk$, bound $\lfloor \beta^2 \rfloor$
Output: signature $sig$ of $m$
 1: $B \leftarrow \begin{bmatrix} g & -f \\ G & -F \end{bmatrix}$
 2: $\hat{B} \leftarrow \mathrm{FFT}(B)$
 3: $G \leftarrow \hat{B} \times \hat{B}^*$
 4: $r \leftarrow \{0,1\}^{320}$   uniformly
 5: $c \leftarrow$ HashToPoint$(r||m, q, n)$
 6: $t \leftarrow \left(-\frac{1}{q}\mathrm{FFT}(c) \odot \mathrm{FFT}(F),\ \frac{1}{q}\mathrm{FFT}(c) \odot \mathrm{FFT}(f)\right)$
 7: do
 8:     do
 9:         $z \leftarrow$ ffSampling_dyntree$_n(t, G)$
10:         $s = (t - z)\hat{B}$
11:     while $||s||^2 > \lfloor \beta^2 \rfloor$
12:     $(s_1, s_2) \leftarrow$ invFFT $(s)$
13:     $s \leftarrow$ Compress $(s_2, 8 \cdot sbytelen - 328)$
14: while $(s = \perp)$
15: return $sig = (r, s)$
```

Algorithm 2 Fast Fourier Sampling (ffSampling)

```
Input: $\mathbf{t} = (t_0, t_1) \in$ FFT $(\mathbb{Q}[x]/(x^n + 1))^2$,
       Gram Matrix $G$
Output: $\mathbf{z} = (z_0, z_1) \in$ FFT $(\mathbb{Z}[x]/(x^n + 1))^2$
 1: if n = 1 then
 2:     $\sigma' \leftarrow$ Normalization$(g_{00})$
 3:     $z_0 \leftarrow$ SamplerZ$(t_0, \sigma')$
 4:     $z_1 \leftarrow$ SamplerZ$(t_1, \sigma')$
 5:     return z = $(z_0, z_1)$
 6: end if
 7: (L, D) $\leftarrow$ LDL*(G)
 8: $l \leftarrow L$
 9: $d_{00}, d_{01} \leftarrow$ split$(\mathrm{D}_{00})$
10: $d_{10}, d_{11} \leftarrow$ split$(\mathrm{D}_{11})$
11: $\mathrm{G}_0 \leftarrow \begin{bmatrix} d_{00} & d_{01} \\ d_{01}^* & d_{00} \end{bmatrix}, \mathrm{G}_1 \leftarrow \begin{bmatrix} d_{10} & d_{11} \\ d_{11}^* & d_{10} \end{bmatrix}$
12: $\mathrm{t}_1 \leftarrow$ split$(t_1)$
13: $\mathrm{z}_1 \leftarrow$ ffSampling_dyntree$_{n/2}$ $(\mathrm{t}_1, \mathrm{G}_1)$
14: $z_1 \leftarrow$ merge$(\mathrm{z}_1)$
15: $t_0' \leftarrow t_0 + (t_1 - z_1) \odot l$
16: $\mathrm{t}_0 \leftarrow$ split$(t_0')$
17: $\mathrm{z}_0 \leftarrow$ ffSampling_dyntree$_{n/2}$ $(\mathrm{t}_0, \mathrm{G}_0)$
18: $z_0 \leftarrow$ merge$(\mathrm{z}_0)$
19: return z = $(z_0, z_1)$
```

Algorithm 3 SamplerZ

```
Input: Floating point values µ,
       $\sigma' \in R$ such that $\sigma' \in [\sigma_{min}, \sigma_{max}]$
Output: An integer $z \in \mathbb{Z}$ sampled from a distribution very
       close to $D_{\mathbb{Z},\mu,\sigma'}$
1: $r \leftarrow \mu - \lfloor \mu \rfloor$
2: $ccs \leftarrow \sigma_{min}/\sigma'$
3: while true do
4:     $z_0 \leftarrow$ GaussianSampler()   ▷ Gaussian sampling
5:     $b \leftarrow$ UniformBits(8) & 0x1
6:     $z \leftarrow b + (2 \cdot b - 1)z_0$
7:     $x \leftarrow \frac{(z-r)^2}{2\sigma'^2} - \frac{z_0^2}{2\sigma_{max}^2}$
8:     if(BerExp(x,ccs) = 1)          ▷ Rejection sampling
9:         return z + $\lfloor \mu \rfloor$
```



```
Zf(sampler) (void *ctx, fpr mu, fpr isigma){
//===================SW=====================
    sampler_context *spc;
    int s, z0, z, b;
    fpr r, dss, ccs, x;
    spc = ctx;
    s = (int)fpr_floor(mu);
    r = fpr_sub(mu, fpr_of(s));
    dss = fpr_half(fpr_sqr(isigma));
    ccs = fpr_mul(isigma, spc->sigma_min);
//===================HW=====================
    for (;;) {
      z0 = Zf(gaussian0_sampler)(&spc->p);
      b = (int)prng_get_u8(&spc->p) & 1;
      z = b + ((b << 1) - 1) * z0;
      x = fpr_mul(
           fpr_sqr(fpr_sub(fpr_of(z), r)),
           dss);
      x = fpr_sub(x,
           fpr_mul(fpr_of(z0 * z0),
           fpr_inv_2sqrsigma0));
      if (BerExp(&spc->p, x, ccs))
          return s + z;
    }
}
```

Fig. 1. Software Implementation of FALCON SamplerZ and its partitioning in previous research [6]

two recursive calls, each of which executes the sampling process on dimensions reduced by half. The detailed steps of the sampling procedure are outlined in Algorithm 2. Starting with the matrix G, the LDL decomposition is applied to split it into a lower triangular matrix L and a diagonal matrix D. The computations within the LDL decomposition rely on complex floating-point operations, highlighting the computational intensity of this step.

D. SamplerZ: Two-Layer Sampling

The `SamplerZ` method consists of two layers of sampling and provides a discrete Gaussian distribution for dynamic parameters μ and σ. The first layer uses `BaseSampler` to generate a sample from a base distribution with constant σ and μ. This sample is then adjusted in the second layer using the `BerExp` algorithm, which applies rejection sampling to ensure that the output conforms to the target Gaussian distribution. During the signature generation, `SamplerZ` calculates variables such as s, r, `ccs`, and `dss` dynamically at runtime, utilizing floating-point operations.

The second layer handles dynamic floating-point calculations, including the use of the `ApproxExp` function to compute the exponential term more efficiently through precomputed tables. The `BerExp` algorithm determines whether the sample from `BaseSampler` is accepted or rejected, iterating the process until a valid sample is obtained. This ensures the statistical accuracy and security of the sampling procedure while maintaining computational efficiency.

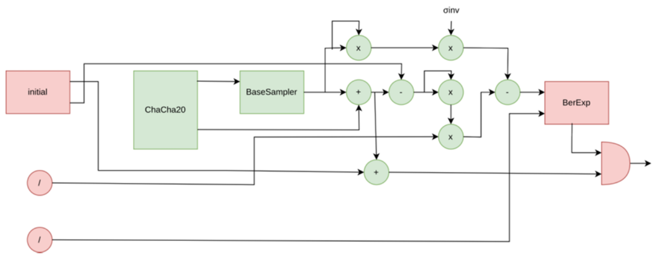


Fig. 2. SamplerZ design

III. Summary of Key Ideas and Novelty

Our main idea is to propose a full hardware implementation of the SamplerZ module (Figure 2). There's no novelty in this implementation.

A. Implementation of Floating-Point Arithmetic Modules

The double-precision floating-point arithmetic units were implemented in compliance with the IEEE-754 standard. These units formed the backbone of the software portion of the Discrete Gaussian Sampling (DGS) function. The floating-point operations you developed included addition, subtraction, multiplication, floor function and conversions between integer and double-precision representations.

B. Generating Parameters `dss`, `ccs`, `r`, and `s`

The parameters `dss`, `ccs`, `r`, and `s` play a crucial role in the Discrete Gaussian Sampling (DGS) process, ensuring statistical accuracy and security of the Gaussian distribution. The computations to generate these parameters were performed using IEEE-754 compliant floating-point arithmetic, ensure numerical precision and reliability, even in edge cases like subnormal numbers or extreme values of μ and `isigma`. The parameters `dss`, `ccs`, `r`, and `s` serve as inputs to the rejection sampling unit, which uses them to generate statistically accurate and secure Gaussian samples for cryptographic operations.

C. ChaCha20

The ChaCha20 cryptographic implementation consists of three main modules working together to implement the ChaCha20 stream cipher for SamplerZ. The top module, *ChaCha20withBuffer*, connects the ChaCha20 core with a buffer system, providing both 8-bit and 64-bit output interfaces while managing data flow between components. The fBuffer module implements a double-buffering system using two 512-bit buffers (`buffer_0` and `buffer_1`) for continuous operation, handling data storage and output formatting, and converting 512-bit blocks into smaller chunks (8-bit or 64-bit). The *ChaCha20* core processes data in 512-bit blocks using the ChaCha20 algorithm with 20 rounds of operations, leveraging a state that includes constant words, key words (256-bit key), nonce words (64 bits), and counter words (64 bits). At the heart of the core is the *QuarterRound* module, which performs addition, XOR, and rotation transformations as the fundamental building blocks of the algorithm.

The design features *double buffering* for seamless operation, allowing one buffer to fill while the other outputs data, thus preventing stalls. It supports flexible output modes with 8-bit and 64-bit data chunks, controlled via a handshaking mechanism (`stb/ack`). The design also incorporates piped operation, where the ChaCha20 core continuously processes data while the buffer handles independent output formatting. The system is controlled by a state machine with distinct states for idle, data fill, and response, ensuring robust separation of control and data paths for efficient and reliable operation.

D. BerExp

The design consists of three main modules: `BerExp` (top module), `CalcSandR`, and `SampleBit`. The top module, `BerExp`, serves as the controller, managing the overall sampling process. It connects input signals for the pseudo-random number generator (PRNG), clock and reset signals, and input parameters `x` and `ccs`, and outputs the sampled bit `b` along with associated control signals. The module orchestrates the pipeline by integrating the `CalcSandR` and `SampleBit` submodules, along with intermediary floating-point operations through `fpr_expm_p63`.

The `CalcSandR` module performs floating-point calculations for sampling by computing the shift amount (`s`) and remainder (`r`). It employs operations such as multiplication with logarithmic constants, truncation, and floating-point format conversions to implement the formula $r = x - s \times \log(2)$. Meanwhile, the `SampleBit` module handles the core Bernoulli sampling through a four-state finite state machine.

IV. Evaluation and Results

A. Resource Utilization and Performance Results

To allow a fair comparision with previous design, we constraint our design to perform the hardware operations in previous research [6]. The hardware implementation of the discrete Gaussian sampling module for FALCON was evaluated on the Xilinx Zynq platform. The post-implementation results demonstrate efficient resource utilization with 9,904 Look-Up Tables (LUTs), corresponding to 18.62% of the available 53,200 LUTs, and 8,663 Flip-Flops (FFs), utilizing 8.14% of the 106,400 available. Additionally, 64 DSP slices were employed, representing 29.09% of the available 220 slices, while 85 Input/Output (IO) pins accounted for 68.00% of the available resources.

Timing analysis showed that all user-specified constraints were met, with a Worst Negative Slack (WNS) of 0.085 ns and a Worst Hold Slack (WHS) of 0.068 ns, ensuring reliable operation. The design achieved a latency of 325 clock cycles at 66 MHz frequency, making it suitable for high-performance cryptographic operations. These results highlight the balance between resource efficiency and computational performance in the proposed hardware implementation.

TABLE I
PERFORMANCE COMPARISON WITH PREVIOUS RESEARCH

Design	Latency (cycles)	Frequency (MHz)	Throughput (sample/sec)
Reference Software [3]	96,747	666	6,890
Previous Design [6]	124	45	362,903
Proposed Design	325	66	203,077

V. DISCUSSIONS

A. Comparisons

Table I compares the performance of the proposed implementation with the reference software and a previous optimized hardware design. The reference software, operating at 666 MHz, exhibits a significant latency of 96,747 cycles and achieves a low throughput of 6,890 samples per second. In contrast, the proposed implementation, operating at a much lower frequency of 66 MHz, achieves a latency of 325 cycles and a throughput of 203,077 samples per second. This translates to a speedup of approximately **29.5×** in throughput compared to the software implementation.

Compared to the previous optimized hardware design, which achieves 362,903 samples per second at 45 MHz with a latency of 124 cycles, the proposed implementation demonstrates a lower throughput due to the absence of performance-specific optimizations such as pipelined arithmetic or specialized memory access techniques. However, the proposed design still offers competitive performance, emphasizing simplicity, correctness, and flexibility.

The significant speedup against the software highlights the advantages of a hardware-based approach, even in the absence of targeted optimizations. The results show that the proposed implementation provides an effective baseline, which can be further improved with additional optimizations such as parallelism or pipeline structures to close the performance gap with optimized designs. This comparison demonstrates the potential of hardware acceleration for discrete Gaussian sampling in cryptographic applications.

B. Limitations

While the proposed implementation demonstrates significant improvements in performance compared to software implementations, it also has certain limitations. One key limitation lies in the design of the arithmetic modules. Each operation in the implementation is handled by a dedicated arithmetic module, as directly translated from the C code. This one-to-one mapping between operations and modules limits the overall efficiency and introduces wasteful storage. Additionally, the handshaking protocol between the different modules can be further optimized. The current protocol for inter-module communication relies on a straightforward mechanism, which introduces latency and can cause delays when modules wait for inputs or acknowledgments. Enhancing the communication protocol to support more efficient data flow and reducing idle times could significantly improve the overall performance and resource utilization of the system. Addressing these limitations would enable the design to fully leverage hardware capabilities and achieve higher throughput and lower latency.

C. Extensions

The optimizations described in prior work provide a clear pathway to improve the performance of our current implementation. One significant enhancement is the efficient handling of large multiplications, particularly in the `BerExp` and `SamplerZ` modules, which are key components of the Gaussian sampling process. The prior work utilizes DSP48E1 blocks in the Xilinx FPGA to split large multiplications into smaller, manageable chunks. For example, a 72-bit and 68-bit multiplication is divided into multiple stages using partial multiplications, effectively fitting the operations into the input size constraints of the DSP blocks. This technique ensures high utilization of DSP resources and reduces latency by enabling parallel execution of sub-operations.

VI. CONCLUSION

In this work, we presented a hardware implementation of discrete Gaussian sampling for FALCON, achieving significant improvements in performance over software implementations. While our design focuses on correctness and serves as a baseline implementation without targeted optimizations, it demonstrates competitive performance and resource efficiency. By incorporating optimizations such as efficient handling of large multiplications, pre-computation of constants, and improved inter-module communication protocols, the proposed design has the potential to achieve state-of-the-art results.

REFERENCES

[1] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[2] T. N. I. of Standards and Technology, "Post-quantum cryptography," https://csrc.nist.gov/projects/post-quantum-cryptography.

[3] P.-A. Fouque *et al.*, "Falcon: Fast-fourier lattice-based compact signatures over ntru," https://falcon-sign.info, 2019.

[4] V. Lyubashevsky *et al.*, "Crystals-dilithium," Submission to the NIST Post-Quantum Cryptography Standardization, 2017.

[5] E. Karabulut, E. Alkim, and A. Aysu, "Efficient, flexible, and constant-time gaussian sampling hardware for lattice cryptography," *IEEE Transactions on Computers*, 2021.

[6] 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