A Hardware-Software Co-Design for the Discrete Gaussian Sampling of FALCON Digital Signature

Emre Karabulut
North Carolina State University
NC, USA
ekarabu@ncsu.edu

Aydin Aysu
North Carolina State University
NC, USA
aaysu@ncsu.edu

***Abstract*—Sampling random values from a discrete Gaussian distribution with high precision is a major and computationally-intensive operation of upcoming or existing cryptographic standards. FALCON is one such algorithm that the National Institute of Standards and Technology chose to standardize as a next-generation, quantum-secure digital signature algorithm. The discrete Gaussian sampling of FALCON has *both flexibility and efficiency* needs—it constitutes 72% of total signature generation in reference software and requires sampling from a variable mean and standard deviation. Unfortunately, there are no prior works on accelerating this complete sampling procedure.**

In this paper, we propose a hardware-software co-design for accelerating FALCON's discrete Gaussian sampling subroutine. The proposed solution handles the flexible computations for setting the variable parameters in software and executes core operations with low latency, parameterized, and custom hardware. The hardware parameterization allows trading off area vs. performance. On a Xilinx SoC FPGA Architecture, the results show that compared to the reference software, our solution can accelerate the sampling up to 9.83× and the full signature scheme by 2.7×. Moreover, we quantified that our optimized multiplier circuits can improve the throughput over a straightforward implementation by 60%.

***Index Terms*—discrete Gaussian sampling, hardware-software co-design, post-quantum cryptography, digital signatures, FPGA.**

I. Introduction

The security of the current large-scale encryption infrastructure is based on the difficulty of solving mathematical problems such as integer factorization [1] and discrete logarithms [2]. Although these problems are conjectured to be hard for classical computers, quantum algorithms are proven to solve them exponentially faster [3], [4]. This poses a serious risk at the core of existing security systems, calling for new cryptographic solutions that can survive the quantum threat.

Recently, the National Institute of Standards and Technology (NIST) has announced algorithms to be standardized for the new, quantum-safe, public-key encryption/establishment and digital signature applications [5]. A major push has just started for transitioning to post-quantum cryptography led by NIST, DHS, Microsoft, Cisco, and Amazon Web Services among others. A vital roadblock to transitioning to practice is efficient implementation, which is especially important for real-time, embedded/edge, and battery-operated devices.

FALCON is one of the algorithms that NIST chose for the post-quantum standards. FALCON uses lattice-based cryptography—a (relatively) new family of cryptographic systems that are based on the short integer solution (SIS) over NTRU lattices [6]. Lattice cryptography includes new types of computations that are absent in earlier cryptographic standards such as RSA and ECDSA. One of these building blocks is discrete Gaussian sampling needed to create the construction of the trapdoors. Having such high-precision Gaussian distributions reduces the signature size and, indeed, FALCON was explicitly chosen for its small signature sizes.

Unfortunately, the signature size savings of discrete Gaussian sampling comes at the expense of computing overhead. This is especially true for FALCON: for the reference software [6] provided in the NIST submission package, sampling can account for 72% of the total signing execution time[1]. Moreover, the sampling in FALCON is different from other lattice-based cryptographic schemes: it requires sampling from variable means and variances. Therefore, a practical implementation of FALCON's sampling procedure requires *both efficiency and flexibility*.

In this work, we propose the first accelerated implementation of FALCON's discrete Gaussian sampling. Specifically, we designed and implemented a hardware-software co-design that can address both the flexibility and efficiency needs. We partition the sampling in such a way that the hardware executes the core operations of sampling (*i.e.,* sampling over the cumulative distribution table, exponent calculation, and rejection sampling) in a configurable manner, while the software performs floating-point divisions. The custom hardware consists of a novel, fully-pipelined and high-throughput datapath. Moreover, the custom hardware includes design-time parameters that change the multiplication pipeline stage, allowing a trade-off between area vs. performance. The software runs on an ARM Cortex-A9 within the Xilinx Zynq SoC FPGA architecture and communicates with the hardware over the AXI bus. The results show that, compared to the NIST reference software [6] compiled on the same platform, our solution can accelerate the sampling up to 9.833×.

The rest of the paper is organized as follows. Section II provides the background on the discrete Gaussian sampling and its special use in FALCON digital signature algorithm.

[1]This obviously changes with respect to the architecture. For implementation result details, please check Section V.

Algorithm 1 FALCON Key Generation Algorithm [6]

Input: A monic polynomial $\phi \in \mathbb{Z}[x]$, a modulus q
Output: A secret key sk and a public key h

$f, g \leftarrow$ Gaussian_CDT_Sampling()
$F, G \leftarrow$ NTRUSolve(f, g, ϕ, q)
$B \leftarrow \begin{bmatrix} g & -f \\ G & F \end{bmatrix}$
$\hat{B} \leftarrow FFT(B)$
$G \leftarrow \hat{B} \times \hat{B}^*$ ▷ $\times$ represents matrix multiplication
$T \leftarrow$ ffLDL$^*(G)$
for each leaf of T **do**
    $leaf.value \leftarrow \sigma / \sqrt{leaf.value}$
$sk \leftarrow (\hat{B}, T)$
$h \leftarrow gf^{-1} mod(q)$
return sk, h

Algorithm 2 FALCON Signature Generation Algorithm [6]

Input: a message m, a secret key sk, a bound β^2
Output: a signature sig of m

$r \leftarrow \{0,1\}^{320}$ uniformly
$c \leftarrow$ HashToPoint $(r||m)$
$t \leftarrow (\frac{-1}{q} FFT(c) \odot FFT(F), \frac{1}{q} FFT(c) \odot FFT(f))$
do ▷ $\odot$ represents FFT multiplication
    do
        $z \leftarrow$ ffSampling (t, T)
        $\mathbf{s} \leftarrow (t - z) \begin{bmatrix} FFT(g) & -FFT(f) \\ FFT(G) & -FFT(F) \end{bmatrix}$
    while $s^2 > [\beta^2]$
    $(s_1, s_2) \leftarrow invFFT(s)$
    $s \leftarrow$ Compress$(s_2, 8 \cdot sbytelen - 328)$
while $s = \perp$
return $sig = (r, s)$

Algorithm 3 BaseSampler [6]

Input: a 72-bit uniformly sampled random number u
Output: a sampled $z_0 \leftarrow D_{\sigma_{base},0}$

$z_0 \leftarrow 0$
for i= 0, ..., 17 **do**
    if $u <$ RCDT[i] **then**
        $z_0 \leftarrow z_0$ + 1
return z_0

Section III describes the target system's architecture and the rationale behind our hardware-software partitioning. Section IV discusses our custom hardware design on the FPGA. Section V presents the implementation results. Section VI explains related aspects and future extensions, and Section VII concludes the paper.

II. Background

This section provides background information about FALCON and its discrete Gaussian sampling sub-routines.

A. Discrete Gaussian Sampling

Gaussian distribution ($D_{\sigma,\mu}$) is determined with a standard deviation σ and a mean (center) μ parameters. The statistical distance from the ideal distribution is determined by a precision parameter λ in the discrete Gaussian distribution. FALCON signature scheme samples its secret coefficients over a discrete Gaussian distribution [6]. Coefficient values range between $-\tau\sigma$ and $\tau\sigma$ where τ is $\sqrt{\lambda \times 2 \times \ln 2}$. The probability of coefficient value x is defined over the distribution and this probability calculated with $\frac{1}{\sigma\sqrt{2\pi}} e^{(x-\mu)^2/2\sigma^2}$.

B. FALCON Post-Quantum Digital Signature Scheme

FALCON is a post-quantum, lattice-based, hash-and-sign signature scheme [6]. FALCON signature scheme has three main steps: key generation, signing, and signature verification. FALCON requires sampling during key generation and signing to generate its secret coefficients; hence, we omit the discussion about the verification procedure in this work. Unlike other NIST PQC finalists [7], [8], FALCON samples its secret coefficients over discrete Gaussian distributions instead of uniform distributions.

Algorithm 1 shows FALCON's key generation that computes the secret key sk and the public key h. The key generation algorithm first samples f and g polynomials over a Gaussian distribution and then generates the secret key by using these two polynomials. Since the key generation algorithm works with constant standard deviation σ and the center μ, the hardware implementation challenges are limited. Therefore, the prior work [9] that presents a hardware Gaussian sampler implementation for FALCON's key generation procedures can be used to this end.

Algorithm 2 illustrates FALCON's signing procedure, which takes in a message m, the signing key sk, and then returns a signature (r and s). The signing algorithm has three main subroutines: HashToPoint, ffSampling, and Compress. In the HashToPoint subroutine, the algorithm first concatenates the message with a uniformly generated polynomial and then hashes it to the polynomial c. Compress subroutine reduces the signature size with a simple encoding mechanism. ffSampling subroutine generates the signature from the secret key and the digested message. FALCON's specification document [6] calls this subroutine the heart of the signature generation.

ffSampling samples signature coefficients over Gaussian distribution where the distrubution σ and μ parameters defined with coefficients of T and t polynomials. Since ffSampling works with dynamic σ and μ parameters, each coefficient of the signature is likely to be sampled over different Gaussian distributions. Although the software's flexibility allows for the implementation of a discrete Gaussian sampler with dynamic σ and μ parameters, efficient hardware acceleration of this sampler is significantly challenging—there was no such hardware to date at the time of this publication.

C. Sampling in FALCON

FALCON has two sampling routines, one for the key generation and the other for the signature generation. The

Algorithm 4 SamplerZ [6]

Input: Floating point values μ, σ' and σ_{min}
Input: Constant $\sigma_{inv} = 1/(2\sigma_{max}^2)$
Output: a sampled coefficient

1: $s \leftarrow \text{ceil}(\mu)$
2: $r \leftarrow \mu - s$
3: $ccs \leftarrow \sigma_{min}/\sigma'$
4: $dss \leftarrow 1/(2 \cdot \sigma'^2)$
5: **while** (1) **do**
6: $z_0 \leftarrow$ **BaseSampler**(u) ▷ u is uniformly random number
7: $b \leftarrow$ OneBitUniformRnd()
8: $z \leftarrow b + (2 \cdot b - 1)z_0$
9: $x \leftarrow ((z - r)^2) \cdot dss - z_0^2 \cdot \sigma_{inv}$
10: **if** (**BerExp**(x, ccs)) **then**
11: return $z + s$

key generation requires a sampling operation over Gaussian distribution with constant σ and μ parameters. By contrast, signature generation requires a Gaussian distribution structure that needs varying σ and μ parameters. Therefore, FALCON employs a sampling strategy that comprises two levels of sampling. The first layer samples a value from the base distribution that has constants σ and μ parameters. The second layer rejects or accepts the sampled value based on the second distribution that satisfies varying σ and μ requirements.

Algorithm 3 shows the first layer of the sampling algorithm used in FALCON, where the input is a 72-bit uniform random number (u) and the output (z_0) is a 5-bit integer from the distribution $D_{\sigma_{base},0}$. This algorithm requires a table that stores pre-computed reverse cumulative distribution values (stored in the RCDT array, line 3). The algorithm first compares a uniform random number against the table and then returns the index of the first entry that is larger. Since all distribution values are pre-calculated and then stored in a table, this algorithm does not need to perform the $exp()$ calculation.

Algorithm 4 presents the $SamplerZ$ method that includes two levels of samplings and provides a discrete distribution for the varying σ and μ. This algorithm calls first $BaseSampler$ at step 6 to sample a non-uniform value over the described Gaussian distribution that has constant σ and μ parameters. Second, it rejects or accepts the sampled value based on the second sampling that is performed by the $BerExp$ algorithm at 10. Note that the $SamplerZ$ function is called many times during FALCON signature scheme with varying inputs (μ and σ'). Therefore, the variables s, r, ccs, and dss are calculated at run-time. These calculations require floating point arithmetic. Although FALCON uses only $BaseSampler$ during the key generation, FALCON calls $SamplerZ$, including both $BaseSampler$ and $BerExp$ algorithms, during the signature generation.

The second layer of the sampling procedure needs to work with dynamic floating-point operations. Algorithm 5 shows the second layer of the sampling procedure. Although this method requires performing the $exp()$ calculation, FALCON reduces

Algorithm 5 BerExp [6]

Input: Floating point values x, $ccs \leq 0$
Output: 1-bit sampling result

1: $s \leftarrow \text{x} \cdot ln(2)^{-1}$
2: $r \leftarrow x - s \cdot ln(2)$
3: **if** $s > 63$ **then**
4: $s \leftarrow 63$
5: $z \leftarrow$(2·ApproxExp(r, ccs) - 1) $>> s$
6: $i \leftarrow 64$
7: **do**
8: $i \leftarrow (i-8)$
9: $urnd \leftarrow$ OneByteUniformRnd()
10: $w \leftarrow urnd - ((z >> i)\&0\text{xFF})$
11: **while** $(w == 0$ && $i > 0)$
12: **if** $w < 0$ **then**
13: return **True**
14: **else**
15: return **False**

Listing 1: FALCON SamplerZ reference implementation [6]. This also shows the partitioning of the reference implementation for HW and SW.

```
1  Zf(sampler) (void *ctx, fpr mu, fpr isigma){
2  //===================SW=====================
3     sampler_context *spc;
4     int s, z0, z, b;
5     fpr r, dss, ccs, x;
6     spc = ctx;
7     s = (int)fpr_floor(mu);
8     r = fpr_sub(mu, fpr_of(s));
9     dss = fpr_half(fpr_sqr(isigma));
10    ccs = fpr_mul(isigma, spc->sigma_min);
11 //===================HW=====================
12    for (;;) {
13       z0 = Zf(gaussian0_sampler)(&spc->p);
14       b = (int)prng_get_u8(&spc->p) & 1;
15       z = b + ((b << 1) - 1) * z0;
16       x = fpr_mul(
17            fpr_sqr(fpr_sub(fpr_of(z), r)),
18            dss);
19       x = fpr_sub(x,
20            fpr_mul(fpr_of(z0 * z0),
21            fpr_inv_2sqrsigma0));
22       if (BerExp(&spc->p, x, ccs))
23           return s + z;
24    }
25 }
```

the computation complexity with $ApproxExp$ implementation (see step 5 in Algorithm 5). $ApproxExp$ approximates the $exp()$ with a pre-computed table. Algorithm 5 returns a one-bit sampling result. This one-bit is indeed the decision bit to accept or reject the $BaseSampler$ output. The $SamplerZ$ re-iterates the steps between 6 and 10 until the first sampled value is accepted by $BerExp$'s output.

III. System Architecture and Hardware-Software Partitioning

FALCON's sampling consists of floating-point addition, multiplication, and division. These operations are expensive

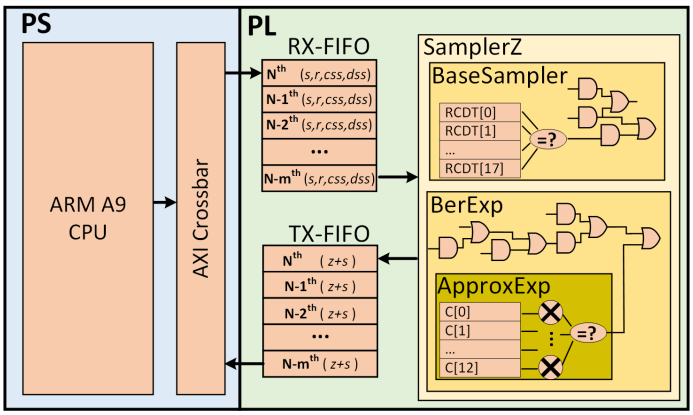


Fig. 1: The proposed architecture for FALCON's sampling operation. The architecture has two partitions that provide a flexible environment where the software and hardware can collaborate and run parallel.

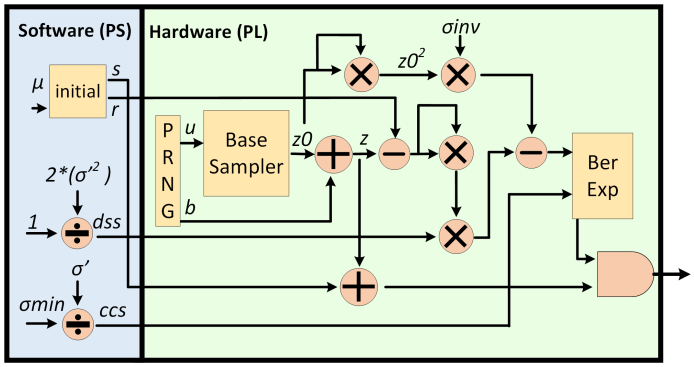


Fig. 2: Software and hardware building blocks of $SamplerZ$ sub-routine. The software (blue) executes floating-point-divisions, while the hardware part (green) performs $BerExp$ and $BaseSampler$.

to implement because of the large operands. Thus, FALCON's C reference implementation (software) demonstrates a sampling mechanism over large integer values. Listing 1 shows FALCON's SamplerZ reference implementation and our partitioning into hardware and software in our system. The sampling requires floating point-based division operations (lines 9 and 10). The reference implementation approximates the operations over large integers r, dss, ccs, and x using **fpr** which is defined with a 64-bit int data type. The algorithm has also varying input that increases hardware implementation complexity and that requires flexibility in the implementation. In addition to the division, the sampling operation needs floating point multiplications in $samplerZ$ (lines 16 and 20) as well as in **BerExp**. There are also memory read operations in **gaussian0_sampler**. Unlike divisions, they are relatively easier to implement on the hardware but these are time-consuming operations in software since the operands are large numbers.

We partitioned the sampling operations into hardware and software workloads to increase throughput while maintaining flexibility. Our partitioning aims to increase parallelism in the sampling operation. The key decider in our hardware-software partitioning is the separation of floating point divisions; while floating point arithmetic is mapped to software (since they already can execute in highly customized ALU datapath in the processor), other computational units are accelerated in custom-designed hardware. The software part consists of floating point-based division and parameter initializations and covers steps between 7 and 10 in Listing 1. The hardware partitioning covers the remaining steps (12 to 23).

Although there are input and output-based data dependencies between the software and hardware groups, there is no intermediate data dependency. The software and hardware operation can thus execute in parallel. As a result, sharing the sampling computation workload among hardware and software can accelerate the sampling operations. While the hardware is working on the first call's while loop, the software can execute s, r, css, and dss for the next call.

We use a Xilinx SoC FPGA that enables running both software and hardware implementations in a single SoC. Figure 1 presents the SoC FPGA architecture and the proposed design's block diagram. Our design utilizes the PS part for performing the operations that requires a flexible architecture such as the floating point-based division, whereas the PL part has dedicated hardware designs to accelerate the operations. Our design has two independent FIFOs that synchronize the software and hardware executions and also enable running in parallel. The software implementation generates the inputs of the hardware accelerator and sends them via RX FIFO. The hardware design sends the output of the $SamplerZ$ algorithm to the software with TX FIFO.

IV. Hardware Design

The proposed hardware design has one accelerator core for $SamplerZ$ and FIFO data paths. $SamplerZ$'s accelerator has three sub-modules: $BaseSampler$, $BerExp$, and $ApproxExp$. Figure 2 illustrates the $SamplerZ$ algorithm's major building blocks and their hardware-software partitions. The software performs initial floating point-based divisions for ccs and dss and parameter initializations for r and s. The hardware accelerator executes $BerExp$, $BaseSampler$, and $ApproxExp$ that require heavy multiplications and memory read operations. For example, the software would sequentially read table entries in the reference $BaseSampler$, and $ApproxExp$ algorithms, while the hardware can read the entire table and perform all comparisons in one clock cycle. The figure also shows $SamplerZ$'s four data paths where the hardware parallelizes the operations.

A. A Half Gaussian Sampling with BaseSampler

We implement the $BaseSampler$ algorithm with a pre-computed reverse cumulative distribution (RCDT) table in hardware, which is carried out in software in FALCON's reference implementation. Figure 1 outlines the $BaseSampler$ implementation in $samplerZ$ module. It has one register file to store RCDT and one comparison circuit. The input of the design is a 72-bit uniform random number and the output is an unsigned integer ranging from 0 to 17.

The design first receives the 72-bit number and then performs a parallel comparison between the input and each entry

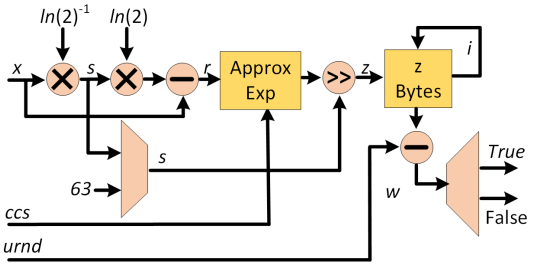


Fig. 3: The block diagram of $BerExp$ sub-routine. It was fully implemented on the hardware.

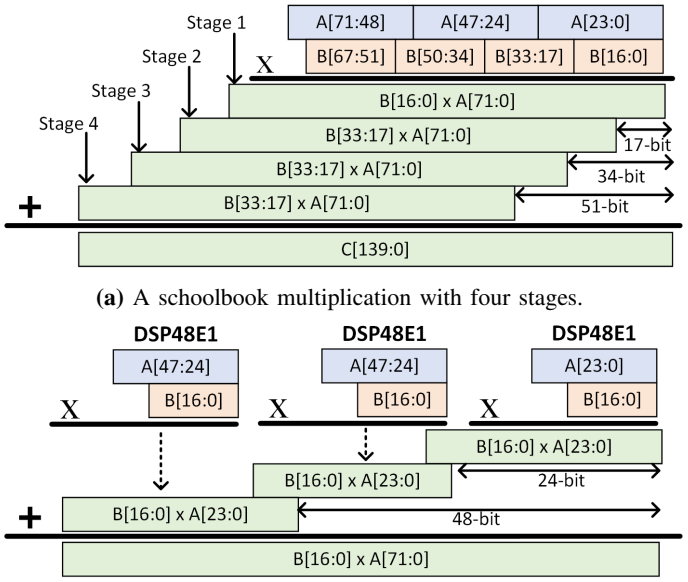
of RCDT. The last step of the design is to count and return the number of entries that are larger than the given input. The software implementation may require multiple cycles to execute a single comparison between the input and one entry of RCDT. By contrast, our hardware implementation can complete the entire table comparison in one cycle.

B. Rejection Sampling with BerExp

The $BerExp$ algorithm returns a single bit and the probability of returning 1 is $css \cdot exp(-x)$. The css value is a floating-point number and defines the scaling factor for each σ' value. The variable x is also a floating-point input that is calculated with the $BaseSampler$'s sampled value. Although the $BerExp$ does not have a floating-point division, it requires floating-point addition and multiplication. FALCON documentation provides the minimum and maximum ranges of the $SamplerZ$ inputs. Therefore, we can simply extend the floating-point numbers with 2^{72} and can work over their integer values with 72-bit precision.

Figure 3 illustrates the $BerExp$ algorithm's computations. Algorithm 5 shows that the input x first goes through a floating-point division at the first step with $ln(2)$ to obtain s. Since $ln(2)$ is a constant value, we calculate its inverse $(ln(2)^{-1})$ and then extend it with 2^{72} at the compile time. After this pre-calculation, the first floating-point division operation simply becomes a 72-bit unsigned multiplication. The first multiplication result is 144 bits. However, we do not reduce this multiplication result to 72 bits before the next multiplication to preserve the precision.

$BerExp$ sampling design also has the $ApproxExp$ module to compute an approximation of $2^{63} \cdot css \cdot exp(-x)$. FALCON's NIST submission package already provides a C implementation of the $ApproxExp$ module. This module computes the approximation over a pre-calculated table (C), like $BaseSampler$. The implementation iteratively performs 64-bit multiplications between table entries and the output of the previous multiplication. The first multiplication happens between the input x and the first table entry. Table C has 13 entries and therefore $ApproxExp$ module initially requires 13 multiplications and then obtains $2^{63} \cdot exp(-x)$. The final step of $ApproxExp$ is a multiplication between ccs and $2^{63} \cdot exp(-x)$.



(a) A schoolbook multiplication with four stages.

(b) One stage multiplication with 3 DSP48E multipliers.

Fig. 4: A multiplication representation for 72-bit and 68-bit unsigned operands. The multiplication is constructed with 17-bit and 24-bit multiplication steps due to the input size constraint of the Xilinx DSP48E1 block.

C. Optimizations in Large Multiplications

FALCON's Gaussian sampling algorithm requires large integer multiplications. The $ApproxExp$ function, for instance, requires 64-bit integer multiplications (14 times) to approximate the exponent operation in FALCON's reference implementation. In addition, FALCON's Gaussian sampling algorithm might call this function more than once to sample a single coefficient based on the number of rejected samples. $BerExp$ and $SamplerZ$ are other steps that require large integer ($\geq$ 72-bit) multiplications, and they may also call several iterations of these multiplications based on the rejection rate.

If FALCON's Gaussian sampling is designed without optimization, dedicated multipliers are separately allocated for each multiplication[2]. This implementation approach results in 116 DSP48E1 utilization because a single DSP48E1 block can multiply at most signed 18-bit with 25-bit variables (or unsigned 17-bit with 24-bit variables) [10]. Hence, large integer multiplication mandates the cascade of several DSP48E1 blocks. Moreover, this unoptimized design operates with low frequencies due to long critical paths caused by the cascaded DSP48E1 blocks. Our proposed design offers a pipelined multiplier structure and optimizes DSP utilization. Therefore, the proposed design operates with a significantly higher frequency.

We first exemplify large integer multiplication with multi-stage 24-bit and 17-bit multiplications using DSPs with maximum utilization. Figure 4a illustrates the product operation of 72-bit and 68-bit operands with the schoolbook multiplication method. The first operand (A) is 72-bit and split into 3 parts with an equal length of 24-bit, while the second operand (B) is split into 17-bit vectors. First, the operand A is multiplied

[2]Simply, the product of A and B is implemented as $A * B$ in HDL.

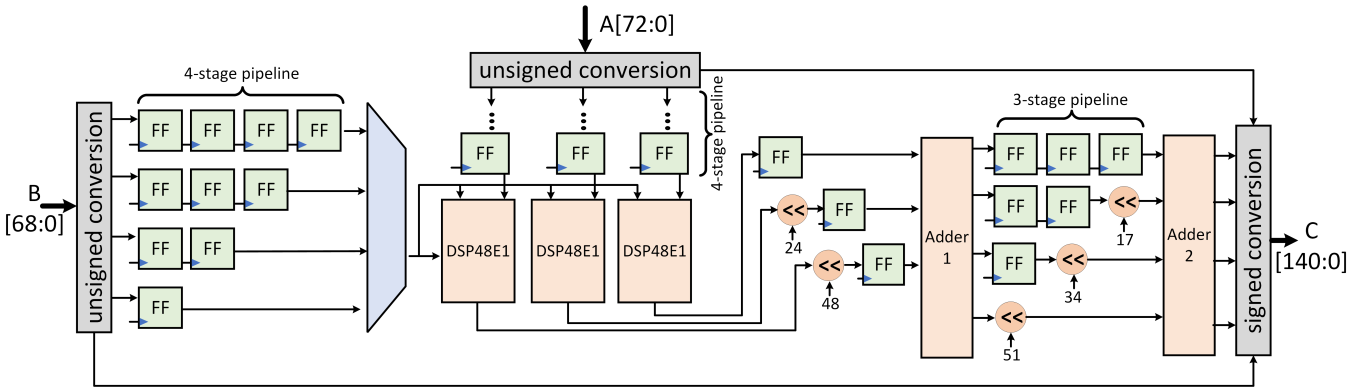


Fig. 5: The proposed multiplier design. This design has multi-stage pipelining that multiplies 73-bit and 69-bit signed operands only with 3 DSP48E1 instead of 16 DSP48E1 blocks. Its latency is 8 cycles and throughput is 1 multiplication per 4 cycles.

by each 17-bit vector of the second operand (B); hence, each stage consists of a 72-bit by 17-bit multiplication. Second, each block multiplication result is shifted left accordingly. The final result is the sum of each stage output.

Figure 4b presents a multiplication block for the first stage of the aforementioned schoolbook multiplication method using three DSP48E1s. Each DSP48E1 shares the operand A with 24-bit fractions, while their other operands are identical, the LSB 17-bit of operand B. Similar to the previous instance, the final result is obtained by first shifting and then summing each multiplication output. The complete hardware, therefore, requires 12 DSP48E1 multiplications, 11 shift operations, and 5 additions to multiply A by B.

Figure 5 shows the proposed pipelined large integer multiplier design. This hardware multiplies two signed operands, one of which can have a maximum size of 73 bits while the other can have a maximum size of 69 bits. The design has a fully-pipelined architecture with 8 stages and thus its latency is 8 cycles but its throughput is one multiplication per 4 cycles. The presented multiplier first converts the two signed operands to unsigned operands before the multiplication operation. Second, it splits the first operand (72-bit) into 3 24-bit groups and the second operand (68-bit) into 4 17-bit groups to fully utilize the DSP48E1 block. Figure 4a shows each stage of the multiplication is executed between the single but different 17-bit vector of B and the second operand A. To sort B's vectors in a cyclic order, the design pipelines each vector with a different number of registers and also selects the corresponding B's vector with a cycle count. For example, the LSB 17-bit of operand B arrives at the DSP48E1 units after the first cycle, while the next 17-bit arrives at the next clock cycle. However, A's vectors are pipelined with the same number of registers since A should remain stable every 4 cycles. Our design utilizes enabled registers and therefore the implementation requires fewer registers for the pipelining. For example, A's vectors are pipelined with one stage of register rather than 4 stages and the register enable signal is activated after every 4 cycles.

The DSP48E1 blocks generate the output of one of the stages in schoolbook multiplication as shown in Figure 4a.

TABLE I: HW utilization of the proposed solution

Task	Module	LUT	FF	DSP	BRAM
Base Design	SamplerZ	2,185	676	116	0
Perf. Opt. Design		3,055	1,960	9	0
Area. Opt. Design		2,670	865	3	0
Sync+Communication	AXI_FIFO	592	632	0	2
Communication	AXI_Periph	405	587	0	0
Reseting	rst_sys	16	23	0	0

Then, the outputs are shifted accordingly before going through the adder circuits. Adder 1 executes the addition operation to execute the model described in Figure 4b, while the second adder is used to perform the addition shown in Figure 4a. The final step of design is the sign extension operation that assigns the corresponding sign to the multiplication product. If these two signed operands' multiplication is implemented with a single asterisk ($*$), the implementation requires 16 DSP48E1 blocks, and that increases DSP48E1 usage by 5.33×.

Our multiplier design also provides design-time flexibility with different area-performance settings. The default configuration uses our performance-optimized design which is described in detail above. If the area-optimization flag is set, the multiplier design works with a single DSP48E1 but, it reduces the throughput and performs one multiplication per 15-cycle. In addition, one multiplication operation in the $samplerZ$ algorithm requires two operands larger than 69-bit but smaller than 73-bit. Therefore, we implemented an additional multiplier that has a 10-stage pipelined design with 3 DSP48E1s.

V. Implementation Results

We used Xilinx Zedboard XC7Z020 from the Zynq-7000 SoC family as our platform. This board has an SoC FPGA, which includes an ARM processor for software execution and a programmable logic part for hardware acceleration. We first designed and functionally verified the proposed $SamplerZ$ and created an AXI-Stream wrapper on it. Second, we hooked the wrapped $SamplerZ$ to FIFOs and Zynq processor as described in Section III. Then, we tested the proposed implementation and measured its performance with FALCON's implementation's test vectors [11].

Table I shows the area utilizations of each IP block as well as their tasks in our block design. We have one baseline design

TABLE II: Performance comparison for different settings

Design	Frequency	Latency (#Cycle)	Throughput (sample/sec)	Improvment
Reference SW [6]	666 MHz	96,747	6,890	-
Base Design	45 MHz	1,691	26,611	**3.861**×
Perf. Opt.	117 MHz	1,726	67,761	**9.833**×
Area Opt.	121 MHz	1,807	66,953	**9.716**×

TABLE III: Theorotical performance result for different settings

Design	Frequency	Latency (#Cycle)	Throughput (sample/sec)	Improvment
Reference SW [6]	666 MHz	96747	6,890	-
Base Design	45 MHz	124	362,903	**52.67**×
Perf. Opt.	117 MHz	212	551,886	**80.09**×
Area Opt.	121 MHz	406	298,029	**43.25**×

and two different optimization settings that offer trade-offs between performance and DSP48E1 usage. The baseline design does not have the proposed multiplier optimization. The first setting is the performance-optimized one that employs three DSP48E1-based multipliers, while the area-optimized design works with single DSP48E1-based multipliers. Table I shows that the base design requires a high number of DSP48E1 units since the base design does not have a dedicated optimization in the multiplier units. Although the base design utilizes $12.88\times$ and $38.66\times$ more DSP48E1 block than the performance-optimized and area-optimized designs, respectively, it uses fewer LUTs and registers, as expected.

Table II presents a performance comparison between FALCON's reference implementation [6] and the proposed designs with different optimization settings. We calculated the improvement result based on the designs' throughputs. The cycle-count results are taken at ARM's clock (666 MHz) frequency. Note that this measurement puts our hardware at a disadvantage: one hardware clock cycle corresponds to 5.67 cycles in the result table. The base design has a lower cycle count than other designs as expected. But its frequency is the lowest one among other designs since it does not have a pipelined architecture. The proposed performance and area-optimized designs outperform FALCON's reference implementation and the base design. However, our area-optimized design almost shows the same performance as our performance-optimized design. This is caused by the AXI components and we later explain the AXI components' impact on the performance with Table III.

Our hardware design consists of two main parts: $SamplerZ$ accelerator core and AXI components. Table II presents a performance result for the entire architecture including the AXI interconnect and AXI FIFO overheads. Therefore, we also present Table III that illustrates only $SamplerZ$ accelerator theoretical performance results without the I/O overhead of carrying the data from the software (PS) side. The results show that a peak throughput improvement of $80.1\times$ is theoretically possible. Since the base design does not have pipelined multipliers, its latency is the best one compared to the other two. However, the performance-optimized design's theoretical throughput is $1.52\times$ and $1.851\times$ more than base designs and area-optimized designs, respectively. Although the base design's throughput is $1.217\times$ better than the area-optimized design, its DSP requirement overhead is $38.66\times$ more than the area-optimized design. Table III also shows that further acceleration may be possible by improving the I/O communication overhead or by porting the design to a more capable FPGA that can execute at a higher operating frequency.

TABLE IV: Hardware design contribution to the performance of $SamplerZ$

Platform	Optimization	Latency (#Cycle)	Our Improvment
ARM Cortex M7 [12]	FPU	664	**18.97** ×
	EMU	3820	**109.14**×
ARM Cortex M4 [12]	FPU	5418	**154.14**×
	EMU	31,744	**906.97**×
Intel® Core® i5-8259U [13]	Reference SW	53	**1.51** ×
Proposed Hardware	Perf Opt	35	-

NIST advocated a more scrutiny evaluation of FALCON's implementation in the Round 2 status report [14] since, unlike other PQC finalists, FALCON employs floating-point arithmetic that might cause implementation errors more likely than other finalists. This NIST call led to a variety of optimizations and benchmarks for FALCON implementation aside from FALCON's reference implementation [12]. There are two types of optimized implementations: one that employs emulated floating-point (EMU) operations and one that uses dedicated floating-point instructions (FPU). FALCON's submission package includes its efficient FPU implementation, which makes use of ARM's dedicated floating-point instructions.

FALCON's FPU optimization can actually be used directly on the software side of our application, with noticeable speed improvements. However, it is essential to compare our hardware partitioning part with its corresponding operations in the FPU and EMU implementations in order to highlight the contribution of our effort. Therefore, we present Table IV to compare the performance of the proposed hardware implementation and the optimized software implementations that perform the hardware's corresponding tasks (see Listing 1 steps 12 to 23). Table IV also presents Intel® Core® i5-8259U performance results for FALCON's reference implementation. Since FPU employs ARM's dedicated floating-point instructions, Intel® Core® i5-8259U does not have performance results for FPU or EMU. To provide a fairer comparison, we compared implementations' performance using cycle count rather than throughput and frequency, taking into account that they are different platforms. We show the cycle count of our performance-optimized design without including the software partitioning part and I/O overhead. We only compare our hardware performance because the FPU is also suitable to work in our software partition part.

Table IV depicts that our novel and efficient hardware implementation significantly improves the performance (up to $906.97\times$) in terms of cycle count. FALCON's sampling procedure requires operations with large operands so ARM Cortex M4 performs the poorest due to its basic structure. Although ARM Cortex M7 is able to benefit from 64-bit floating-point instructions, our hardware implementation executes the

TABLE V: Improvement in performance

Operation	Reference SW [6] Cycle Count	Our Design Cycle Count	Improvment
SamplerZ	96,747	9,838	**89%**
GaussianSampling	99,068,928	10,074,112	**89%**
ffSampling	126,157,824	41,632,081	**67%**
FALCON Signature Generation	136,234,371	50,406,717	**63%**

same operation 18.97× and 109.14× faster. Our hardware even outperforms the modern Intel® Core® i5-8259U CPU by showing 1.51× better performance even though our hardware is implemented on a low-power and modest SoC FPGA. We also note that FALCON's Gaussian sampler execution is not constant-time for its software implementations. Therefore, the cycle count numbers of the software execution might slightly vary for different runs.

Next, we quantify the impact of the improvement on the whole digital signature scheme, not just on the discrete Gaussian distribution. Table V compares the performance between FALCON's reference implementation vs. our performance-optimized design for discrete Gaussian sampling. We first profiled the $SamplerZ$ function by running reference implementation and then run the same function by enabling our accelerator. The result shows that our accelerator improves the $SamplerZ$ execution time by 89%. Since the Gaussian sampler is called $2n$ times during the ffSampling, our design decreased the Gaussian sampler's execution cycle count from 99M to 10M cycles. FALCON's signature generation heavily depends on the performance of the Gaussian sampler. During the signing procedure, FALCON spends 72% of execution time on the Gaussian sampling [15]. Therefore, our design improves the entire signature generation by 63%, from 136M cycles down to 50M.

Table VI presents a comparison between the proposed design and prior Gaussian samplers. There are earlier sampler designs for BLISS [16], [17], LP [16], [18], [19], FrodoKEM [20] and qTESLA [21]. These hardware implementations are fixed for a single σ, μ parameter setting; hence, they **cannot** support FALCON. An earlier FALCON implementation [9] proposed a 'design-time' flexible hardware for σ, μ parameters—this hardware cannot support run-time flexibility and thus is limited to FALCON key generation (i.e., cannot support FALCON signature generation). Other FALCON implementations include FALCON's verification steps [22], which excludes Gaussian sampling, and SIMD acceleration on software [23] [3]. Likewise, another prior work passes FALCON software implementation through an HLS tool but does not provide a performance or area profiling for the Gaussian sampler sub-routine [24]. These results show that, despite our hardware optimizations, FALCON's sampling needs still incur more time and area overheads compared to other lattice-based cryptosystems. FALCON has also a GPU implementation[4] [25]. However, this work does not provide a performance result for FALCON's Gaussian sampler. As a result, we cannot provide a discussion about comparing our work with GPU platforms.

[3]Unfortunately, the discrete Gaussian sampling sub-routine latency is not provided in this work; hence, a comparison is infeasible

[4]This paper has not been officially published but posted in IACR.

TABLE VI: Comparison with prior works

Work	FALCON Support	σ/λ	Platform	LUT/FF/ BRAM	F_{Max} (MHz)	Cyc Cnt
This Work Area Opt.	**Full Support**	**varying**	XC7Z020	3683/2107/2	121	1807
[9]	**KeyGen only**	2/53	Virtex-7	151/7/0	322	1
		$\sqrt{5}$/200		455/8/0	317	1
[21]	**No**	8.5/64	Artix-7	907/812/3	115	111
[26]		8.5/64	Artix-7	511/343/0	353	1
[16]	**No**	3.33/64	Virtex-6	112/19/0	297	5
[18]		3.33/90	Virtex-5	43/33/1	259	3
[19]		3.33/80	Virtex-6	863/6/0	61	1
[27]	**No**	215/128	Spartan-6	928/1121/0	129	8
[28]	**No**	4.41/112	Spartan-6	426/123/1	102	8
[29]		4.41/112	Spartan-6	463/45/0	80	30

VI. Discussions

Implementation Security. We do not cover implementation attacks and associated defenses in this work. There are various attack vectors including fault injection attacks, power/EM side-channel attacks, microarchitectural attacks, acoustic/photonic side-channel attacks, and cold-boot attacks, among others. These attacks have to be evaluated and related defenses could be added on top of our solution. Note that we propose the first-ever hardware acceleration of FALCON sampling procedure. The natural steps in this line of work are to first develop hardware/software solutions for algorithms and then to consider such attacks in follow on studies. This is exemplified in many previous works, including the hardware design of discrete Gaussian samplers without implementation security [30], hardware design of lightweight cryptography algorithms without implementation security [31], and software design for fully homomorphic encryption without implementation security [32], among others.

Performance Comparison on Lower-End FPGA Devices. If ported on lower-end devices that contain an embedded microcontroller without floating-point hardware support, the software side can take longer. In such cases, the floating point arithmetic (i.e., the entire sampling process) can be moved to FPGA for acceleration, provided that the FPGA contains sufficient space. This work demonstrates a novel hardware-software co-design method for accelerating FALCON's discrete Gaussian sampling sub-routine. This work does not aim to optimize and accelerate the AXI components because we argue that this optimization effort does not contribute to the novelty of the proposed method.

VII. Conclusions

FALCON is one of the algorithms that NIST chose yet its implementation has been omitted in prior work. Implementing FALCON efficiently requires accelerating its discrete Gaussian sampling algorithm, which is non-trivial because it includes different components compared to other Gaussian samplers used in lattice cryptography. This paper demonstrates that a

hardware-software co-design method is suitable for addressing both the efficiency and flexibility needs used in FALCON. Our solution accelerates the reference sampling software by 89%, which corresponds to a total improvement of 63% for the signature generation.

REFERENCES

[1] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

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

[3] P. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Review*, vol. 41, no. 2, pp. 303–332, 1999.

[4] J. Proos et al., "Shor's discrete logarithm quantum algorithm for elliptic curves," *Quantum Info. Comput.*, vol. 3, no. 4, pp. 317–344, Jul. 2003.

[5] The National Institute of Standards and Technology, "Post-quantum cryptography pqc," https://csrc.nist.gov/projects/post-quantum-cryptography.

[6] P. A. Fouque et al., "Falcon: Fast-fourier lattice-based compact signatures over NTRU."

[7] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, D. Stehlé, and S. Bai, "Crystals-dilithium," *Submission to the NIST Post-Quantum Cryptography Standardization [NIS]*, 2017.

[8] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, "The sphincs+ signature framework," Cryptology ePrint Archive, Paper 2019/1086, 2019, https://eprint.iacr.org/2019/1086. [Online]. Available: https://eprint.iacr.org/2019/1086

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

[10] *7 Series DSP48E1 Slice*, Xilinx Inc, 3 2018, v1.10.

[11] T. Prest, "falcon.py," 12 2022. [Online]. Available: https://github.com/tprest/falcon.py.git

[12] J. Howe and B. Westerbaan, "Benchmarking and analysing nist pqc lattice-based signature scheme standards on the arm cortex m7."

[13] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, Z. Zhang *et al.*, "Fast-fourier lattice-based compact signatures over ntru," https://falcon-sign.info/.

[14] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, J. Kelsey, Y.-K. Liu, C. Miller, D. Moody, R. Peralta *et al.*, "Status report on the second round of the nist post-quantum cryptography standardization process," *US Department of Commerce, NIST*, vol. 2, 2020.

[15] T. Oder, J. Speith, K. Höltgen, and T. Güneysu, "Towards practical microcontroller implementation of the signature scheme falcon," in *International Conference on Post-Quantum Cryptography*. Springer, 2019, pp. 65–80.

[16] J. Howe, A. Khalid, C. Rafferty, F. Regazzoni, and M. O'Neill, "On practical discrete gaussian samplers for lattice-based cryptography," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 322–334, 2016.

[17] T. Pöppelmann and T. Güneysu, "Area optimization of lightweight lattice-based encryption on reconfigurable hardware," in *2014 IEEE international symposium on circuits and systems (ISCAS)*. IEEE, 2014, pp. 2796–2799.

[18] C. Du and G. Bai, "Towards efficient discrete gaussian sampling for lattice-based cryptography," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2015, pp. 1–6.

[19] T. Pöppelmann and T. Güneysu, "Towards practical lattice-based public-key encryption on reconfigurable hardware," in *International Conference on Selected Areas in Cryptography*. Springer, 2013, pp. 68–85.

[20] J. Howe, T. Oder, M. Krausz, and T. Güneysu, "Standard lattice-based key encapsulation on embedded devices," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 372–393, 2018.

[21] S. Tian, W. Wang, and J. Szefer, "Merge-exchange sort based discrete gaussian sampler with fixed memory access pattern," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 126–134.

[22] P. Karl, J. Schupp, T. Fritzmann, and G. Sigl, "Post-quantum signatures on risc-v with hardware acceleration," Cryptology ePrint Archive, Paper 2022/538, 2022, https://eprint.iacr.org/2022/538. [Online]. Available: https://eprint.iacr.org/2022/538

[23] K. Kiningham, P. Levis, M. Anderson, D. Boneh, M. Horowitz, and M. Shih, "Falcon — a flexible architecture for accelerating cryptography," in *2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, 2019, pp. 136–144.

[24] D. Soni, K. Basu, M. Nabeel, N. Aaraj, M. Manzano, R. Karri, D. Soni, K. Basu, M. Nabeel, N. Aaraj *et al.*, "Falcon," *Hardware Architectures for Post-Quantum Digital Signature Schemes*, pp. 31–41, 2021.

[25] W.-K. Lee, R. K. Zhao, R. Steinfeld, A. Sakzad, and S. O. Hwang, "High throughput lattice-based signatures on gpus: Comparing falcon and mitaka," *Cryptology ePrint Archive*, 2023.

[26] L. Kong, R. Liu *et al.*, "High-performance constant-time discrete gaussian sampling," *IEEE Transactions on Computers*, 2020.

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

[28] C. Zhang, Z. Liu, Y. Chen, J. Lu, and D. Liu, "A flexible and generic gaussian sampler with power side-channel countermeasures for quantum-secure internet of things," *IEEE Internet of Things Journal*, 2020.

[29] D. Liu, C. Zhang, H. Lin, Y. Chen, and M. Zhang, "A resource-efficient and side-channel secure hardware implementation of ring-lwe cryptographic processor," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 4, pp. 1474–1483, 2018.

[30] R. Agrawal, L. Bu, and M. A. Kinsy, "A post-quantum secure discrete gaussian noise sampler," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, pp. 295–304.

[31] A. A. Ayoub and M. D. Aagaard, "Application-specific instruction set architecture for an ultralight hardware security module," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, pp. 69–79.

[32] T. Morshed, M. M. A. Aziz, and N. Mohammed, "Cpu and gpu accelerated fully homomorphic encryption," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020, pp. 142–153.