LEAF: Lightweight and Efficient Hardware Accelerator for Signature Verification of FALCON

Samuel Coulon[1], Jinjun Xiong[2] and Jiafeng Xie[1]
[1]: Department of Electrical and Computer Engineering, Villanova University, Villanova, PA 19087
[2]: Department of Computer Science and Engineering, University at Buffalo, NY 14260
[1]: Email:{scoulon, jiafeng.xie}@villanova.edu; [2]: Email: jinjun@buffalo.edu.

***Abstract*—Along with the National Institute of Standards and Technology (NIST) post-quantum cryptography (PQC) standardization process, efficient hardware acceleration for PQC has become a priority. Among the NIST-selected PQC digital signature schemes, FALCON shows great promise due to its compact key sizes and efficient Signature Verification procedure. However, FALCON is regarded as highly computationally complex, and as a result, few works for hardware acceleration of FALCON can be found in the literature, where the few existing ones only target high-performance. To fill the gap, this paper presents a Lightweight and Efficient hardware accelerator for the Signature Verification portion of FALCON (LEAF), specifically for resource-constrained applications. We propose an efficient design strategy, including a novel data dependence flow, to maximize the utilization of very small resources for all arithmetic procedures. Then, the proposed full-hardware LEAF is built, containing an ultra-lightweight number theoretical transform (NTT) core with a novel twiddle factor access pattern. Finally, we conduct a thorough evaluation to demonstrate the efficiency of LEAF. To the best of our knowledge, this is the first lightweight and meanwhile most resource-efficient FALCON Signature Verification full-hardware accelerator in the literature, offering 65% and 66% less aggregate resource usage and achieving 24% and 14% less equivalent area-time product (eATP), compared to the state-of-the-art for FALCON-512 and FALCON-1024, respectively. We hope that this work can spur further research in the field.**

***Index Terms*—FALCON scheme, field-programmable gate array (FPGA), post-quantum cryptography (PQC), hardware accelerator, lightweight and efficient design, Signature Verification.**

I. INTRODUCTION

Given the looming threat of future quantum computers to current cryptographic systems, new quantum-resistant algorithms (namely post-quantum cryptography (PQC)) are being developed in response [1], [2]. The National Institute of Standards and Technology (NIST) has concluded its initial PQC standardization process, and has selected four algorithms for standardization [3]. Along with this trend, many related hardware design works have also been released recently [4].

Among the four NIST-selected PQC algorithms [3], for FALCON, more attention should be placed on the hardware design [4], [5]. Due to FALCON's highly sophisticated algorithmic operations, not many related hardware works have been reported [6]–[8]. Nevertheless, as FALCON possesses unique features such as fast Signature Verification and compact key size (among all NIST-selected digital signature schemes) [5], the full-hardware designed FALCON will be very useful for practical usage, especially for applications where fast signature verification and compact key size are highly preferred such as the resource-constrained Internet-of-Things (IoT) [9].

Prior Works. In general, there exist three types of hardware implementation works for PQC schemes: (i) the high-level synthesis (HLS)-based ones, (ii) the hardware-software co-design ones, and (iii) the full-hardware implemented designs [4]. As for FALCON, in an early work, HLS was used to implement FALCON's Signature Verification [10]. Another HLS-based design was given in [11], but with very large area usage. Several hardware-software co-design works for FALCON can be seen in [6], [12]–[14] and incur relatively long processing time. [12] presented a hardware-software co-designed sampler, [13] reported the Signature Generation phase of FALCON, [6] gave a hardware-software co-design Fast Fourier Transform (FFT) for FALCON, and [14] reported the hardware-software co-designed Signature Verification. Full-hardware implemented designs offer the best performance upside, but at the cost of complicated control flow and longer development time. The existing full-hardware works focus mostly on FALCON's individual components, with one implements the full Signature Verification: (i) Signature Verification of FALCON in [8]; (ii) FFT accelerator in [15]; (iii) schoolbook-based polynomial multiplication designs [16], [17]; (iv) residue number system decomposition in [18] and large integer extended greatest common divisor for FALCON in [7]; (v) hardware floating-point multiplication and masking [19]; (vi) hardware-accelerated Signature Generation of FALCON [20]. Other indirectly related works are the number theoretic transform (NTT) designs that use the same parameters as FALCON, including (but not limited to) [21]–[28].

Motivation. Overall, the hardware acceleration research for FALCON is still not sufficient, especially on the full-hardware design side where most of the existing works focus on individual components and very few on a complete operational phase like Signature Verification [8]. Meanwhile, according to the overview of the existing research activities, most of the existing works are targeting high-speed applications [8], [17], and few efforts have been made specifically on lightweight design for resource-constrained environments. Taking FALCON's unique features into consideration, i.e., compact key sizes and fast verification [5], a lightweight full-hardware accelerated

*: This paper is an updated version of the paper that was accepted at the 2025 International Conference on Computer-Aided Design (ICCAD'25).

FALCON will be very promising for potential applications. Nevertheless, related research in this area is still severely lacking, and presents great opportunity for development.

Major Contributions. Following the above discussions, we aim to deliver a **L**ightweight and **E**fficient hardware **A**ccelerator for the Signature Verification of **F**ALCON (LEAF), specifically targeting resource-constrained devices. The key contributions of these efforts include:

- A lightweight yet efficient design strategy is proposed, including a novel data dependency flow for FALCON Signature Verification, to efficiently schedule all procedures and maximize utilization of just a single DSP for all arithmetic operations.
- A lightweight, efficient, full-hardware accelerator for FALCON Signature Verification (LEAF) is built, including an ultra-lightweight NTT-based polynomial multiplication core with minimized address management overhead.
- A detailed performance evaluation is provided, covering both the NTT-based polynomial multiplication core and the accelerator, to showcase the proposed LEAF's reduced resource usage and improved area-time complexities compared to the state-of-the-art designs. To our best knowledge, this is the first full-hardware-implemented lightweight FALCON Signature Verification accelerator.

The rest of the paper is organized as follows. Section II provides preliminary information. Section III presents the proposed design strategy and data dependence flow. Section IV gives the details of the proposed LEAF. Section V delivers the evaluation and Section VI concludes the paper.

II. PRELIMINARIES

A. Notations

We have used the following notations throughout the paper. FALCON Signature Verification is performed over Ring $\mathbb{Z}_q[x]/(x^n+1)$ with $q = 12,289$ as the prime modulus and $n = 512/1,024$ as the polynomial degree (corresponds for security levels I and V, respectively) [5]. Polynomials are indicated as bold ($\boldsymbol{v}$) with coefficients indexed by subscript ($\boldsymbol{v}_0 ... \boldsymbol{v}_{n-1}$). Coefficient point-wise multiplication is $\boldsymbol{v} \otimes \boldsymbol{w}$. For NTT operations, ψ is a $2n$-th root of unity, where $\psi^{2n} = 1 \bmod q$ and $\psi^i \neq 1 \bmod q$ for all $i = 1, ..., 2n-1$. Clock Cycles is often abbreviated as "CCs". Other notations can be seen in FALCON's NIST submission package [5] and/or in specific places in this manuscript.

B. FALCON *Signature Verification*

Signature Verification is one of the three major phases in the FALCON digital signature scheme [5]. The procedure is shown in Algorithm 1, and additional parameters are shown in Table I. The inputs of Signature Verification include a Signed Message sm and a Public Key pk. sm contains a Message msg, a random $nonce$, and a Signature sig. There is also an acceptance bound $\lfloor \beta^2 \rfloor$, which is used to verify that the signature is short enough (and thus is valid). Once the bytes of sm are parsed into sig, msg, and $nonce$ (done off-chip), the msg and $nonce$ are hashed into $\boldsymbol{c}$ using `SHAKE256`. The pk is decoded into $\boldsymbol{h}$, and sig is decompressed into $\boldsymbol{s^{(2)}}$. Both the `Decode` and `Decomp` functions operate over a stream of bytes and extract coefficients in $\mathbb{Z}_q$. Next, the decoded $\boldsymbol{h}$ and $\boldsymbol{s^{(2)}}$ are converted to the NTT domain. Then their coefficients are multiplied together point-wise, and the resultant polynomial is converted back with INTT to produce the polynomial multiplication result. Next, the resultant polynomial is subtracted from $\boldsymbol{c}$ to get $\boldsymbol{s^{(1)}}$. Lastly, all coefficients $\boldsymbol{s_i^{(1)}}$ and $\boldsymbol{s_i^{(2)}}$ are squared and accumulated to find the L2 norm, which is compared with bound $\lfloor \beta^2 \rfloor$ to determine if the signature is $valid$ or not.

Algorithm 1: FALCON Signature Verification [5]

Input : sm, pk, $\lfloor \beta^2 \rfloor$
Output: Signature $valid$ or $invalid$
Initialization
1 sig, msg, $nonce \leftarrow$ `parse`(sm)
Procedure
2 $\boldsymbol{c} \leftarrow$ `HashToPoint`($msg \,\|\, nonce$)
3 $\boldsymbol{h} \leftarrow$ `Decode`(pk)
4 $\boldsymbol{s^{(2)}} \leftarrow$ `Decomp`(sig)
5 $\boldsymbol{s^{(1)}} \leftarrow \boldsymbol{c} -$ `INTT`(`NTT`($\boldsymbol{h}$) $\otimes$ `NTT`($\boldsymbol{s^{(2)}}$))
6 **if** $\|(\boldsymbol{s^{(1)}}, \boldsymbol{s^{(2)}})\|^2 \leq \lfloor \beta^2 \rfloor$ **then**
7 **Return** $valid$
8 **else**
9 **Return** $invalid$
10 **end**

TABLE I
FALCON SIGNATURE PARAMETERS

n	512	1,024
pk size (bytes)	896	1,792
sig size (bytes)	617	1,232
msg size (bytes)	1 to 1,300	1 to 2,850
$nonce$ size (bytes)	40	40
$\lfloor \beta^2 \rfloor$	34,034,726	70,265,242

Note that Line 5 of Algorithm 1 was $\boldsymbol{s^{(1)}} \leftarrow \boldsymbol{c} - \boldsymbol{h} \times \boldsymbol{s^{(2)}}$ in the original documentation of [5], but we have changed it into the format of applying NTT-based operations.

III. PROPOSED DESIGN STRATEGY

We target low resource utilization for implementation on resource-constrained devices. As such, we need to select arithmetic and architectural methods that are friendly for lightweight hardware implementation, while ensuring performance is still adequate. We employ the following two strategies to accomplish this: (i) We have carefully selected algorithmic primitives that are suitable for lightweight resource utilization, such as an NTT algorithm with ultra-simple addressing overhead, and a modular reduction algorithm that is tailored for modulus $q = 12,289$. In terms of architecture, we seek to use few expensive resources like DSPs, so our NTT uses only 1 butterfly core with 1 DSP. (ii) We have created an optimized operation schedule to streamline processing and maximize utilization for the single DSP in our design, avoid timing loss, and effectively utilize resources.

A. Selected Algorithmic Primitives

Modular Reduction. The two commonly chosen algorithms for modular reduction are the Montgomery [29] and Barrett [30] methods. The Montgomery method is used in the FALCON reference code [5]. It is known to be favorable for applications with many moduli, but less efficient for individual multiplications with a single modulus [31], so many designs use the Barrett method [32]. We opt for a specialized reduction algorithm from [22] which targets $q = 12,289$. Their algorithm exploits properties of the specific modulus and recursively applies the property $2^{14} \equiv 2^{12} - 1 \bmod q$ to replace expensive multiplications with accumulations, shown in Algorithm 2. In our actual experiments, the method from [22] resulted in about the same resource usage, but required only 4 (instead of 6) pipeline stages to achieve the same maximum frequency, compared with our implementation adopted from the Barrett reduction of [32]. This improvement eliminates the need for additional pipeline registers in our arithmetic core, resulting in lower overall area consumption. Therefore, we select the method from [22] for modular reduction.

Algorithm 2: Modular Reduction for $q = 12,289$ [22]

Input : $B = A_1 \times A_2$ $(A_1, A_2 \in \mathbb{Z}_q)$
Output: $B \bmod q$
Procedure
1 $C \leftarrow B[27:26] + B[25:24] + B[23:22] + B[21:20] + B[19:18] + B[17:16] + B[15:14]$
2 $D \leftarrow B[27:26] + B[27:24] + B[27:22] + B[27:20] + B[27:18] + B[27:16] + B[27:14]$
3 $E \leftarrow C[4] + C[3:2] + C[1:0]$
4 $F \leftarrow (2^{12}(E[2] + E[1:0])) - (E[2] + C[4] + C[4:2])$
5 $G \leftarrow (F + B[13:0]) \bmod q$
6 $H \leftarrow (G - D) \bmod q$
7 **Return** H

NTT. We employ the NTT to accelerate the expensive polynomial multiplication operations of Algorithm 1. Traditionally, the Cooley-Tukey (CT) [33] and Gentleman-Sande (GS) [34] algorithms are used for the forward NTT and inverse NTT (INTT) operations, respectively. Notably, these algorithms can be performed in-place, so total memory usage is lower. The main drawback is CT and GS require complex access patterns that vary between NTT stages, requiring careful address management to avoid memory conflicts. A third option, the Constant-Geometry (CG) algorithm [35], maintains the same memory access pattern between stages of the NTT computation, resulting in simplified address management. The trade-off is that CG is an out-of-place algorithm, so additional memory is required for handling operands during the transform, often referred to as the ping-pong method.

More recently, the authors of [36] developed a unified CG-based algorithm (Algorithm 3) where the forward NTT and inverse INTT access patterns are the same, yielding optimally low-complexity access pattern management. They have also innovatively merged the pre- and post-processing steps into the butterfly. One further benefit of the out-of-place structure is that the bit-reversal re-order step (Line 1 of Algorithm 3) can easily be implemented by simply bit-reversing the read addresses of the first NTT stage, so it does not require additional cycles, as suffered by the designs of [22], [24].

To further offset the increased memory usage, we exploit the property of $\psi^{-(n-i)} = q - \psi^i$ to dynamically compute INTT twiddle factors, so we only store the forward NTT twiddle factors, reducing twiddle factor ROM usage by 50%.

Following the above discussion, we hypothesize that the small increase in memory usage (0.5 more BRAM) for CG will be offset by the significant reduction in logical complexity for address management and reduced twiddle ROM usage.

Hashing. We utilize the Keccak hashing core to implement the `HashToPoint/SHAKE256` function, hashing $msg \parallel nonce$ to $\boldsymbol{c}$ (Line 1 of Algorithm 1). In particular, we use the "mid-range" core which can ingest and hash 136 Bytes every 843 clock cycles (CCs) [37]. Before $\boldsymbol{c}$ must be ready, the earlier operations require 8,319 and 17,924 CCs for $n = 512$ and $n = 1,024$ respectively. With a 40-byte $nonce$, this translates to a maximum msg size of around 1.30kB and 2.85kB, before hashing $msg \parallel nonce$ becomes a bottleneck.

Algorithm 3: Unified CG NTT/INTT [36]

Input : $\boldsymbol{A}$, $n = 512/1,024$, $q = 12,289$, $inverse$, ψ^i, ψ^{-i}
Output: NTT($\boldsymbol{A}$) or INTT($\boldsymbol{A}$)
Procedure
1 $\boldsymbol{A} \leftarrow \mathtt{BRA}(\boldsymbol{A})$
2 **for** $stage = 1$ to $\log_2(n)$ **do**
3 **for** $cc = 0$ to $\frac{n}{2} - 1$ **do**
4 **if** $inverse = 0$ **then**
5 $k_1 \leftarrow \lfloor cc/2^{\log_2(n)-stage} \rfloor \times 2^{\log_2(n)-stage}$
6 $k_2 \leftarrow n/2^{stage}$
7 $k \leftarrow 2k_1 + k_2 \bmod n$
8 $\overline{\boldsymbol{A}}_{cc} \leftarrow \boldsymbol{A}_{2cc} + (\boldsymbol{A}_{2cc+1} \times \psi^k) \bmod q$
9 $\overline{\boldsymbol{A}}_{cc+n/2} \leftarrow \boldsymbol{A}_{2cc} - (\boldsymbol{A}_{2cc+1} \times \psi^k) \bmod q$
10 **else**
11 $k_1 \leftarrow \lfloor \mathtt{BR}(cc)/2^{stage-1} \rfloor \times 2^{stage-1}$
12 $k_2 \leftarrow 2^{stage-1}$
13 $k \leftarrow 2k_1 + k_2 \bmod n$
14 $\overline{\boldsymbol{A}}_{cc} \leftarrow (\boldsymbol{A}_{2cc} + \boldsymbol{A}_{2cc+1}) \times \frac{1}{2} \bmod q$
15 $\overline{\boldsymbol{A}}_{cc+n/2} \leftarrow (\boldsymbol{A}_{2cc} - \boldsymbol{A}_{2cc+1}) \times \psi^{-k} \times \frac{1}{2} \bmod q$
16 **end**
17 **end**
18 $\boldsymbol{A} \leftarrow \overline{\boldsymbol{A}}$
19 **end**
20 **Return** $\boldsymbol{A}$
BRA($\cdot$): bit-reversal of array order
BR($\cdot$): bit-reversal of number

B. Proposed Data Dependency Flow

To effectively design a lightweight accelerator while maintaining adequate speed, we need to carefully plan how the operations listed in Algorithm 1 are scheduled using the limited set of components, namely the single instantiated DSP. The proposed data dependency flow and related operation schedule are shown in Fig. 1. Analyzing the operations of Lines 2-6 of Algorithm 1, Lines 2-4 have no inter-dependencies and can be initiated in parallel. Next, we note that Line 6 involves the squaring and accumulation of the $\boldsymbol{s_i^{(2)}}$ resulting from Line 4. The squaring and accumulation of $\boldsymbol{s_i^{(2)}}$ must be performed before Line 5 (since the $\boldsymbol{s_i^{(2)}}$ coefficients will be transformed with the NTT), so we also begin the operations of Line 6 in

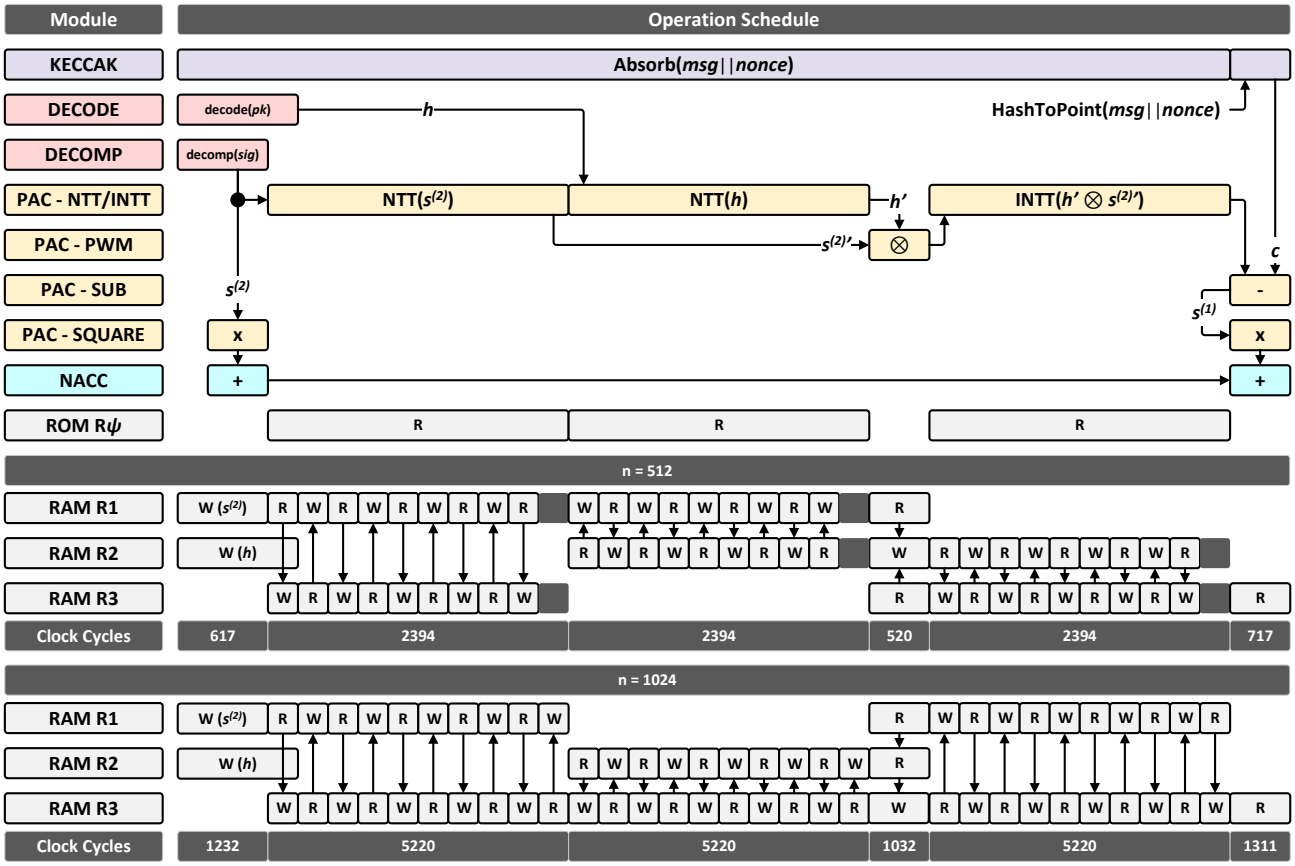


Fig. 1. Proposed data dependency flow and operation schedule for FALCON's Signature Verification.
Note: the lower half part of the scheduling and timing match the proposed LEAF in Section IV (R: read; W: write).

parallel as $s_i^{(2)}$ values become available, using the DSP. The `Decomp` function on Line 4 should finish before the operations of Lines 2-3, so we start with the transform of NTT($s^{(2)}$) next. Once this operation finishes, the `Decode` operation of Line 3 will be complete, so we next perform the transform of NTT(h). Once this is done, we can multiply the coefficients NTT($s_i^{(2)}$) and NTT(h_i) point-wise, followed by the inverse transform INTT of the result. The two forward NTTs, the point-wise multiplication (PWM), and the INTT occur back-to-back to maximize utilization of the DSP. Next, assuming the msg has been fully absorbed by the Keccak core, we perform the `HashToPoint` on Line 2 to produce coefficients c_i, and subtract the polynomial multiplication results, producing $s^{(1)}$ and completing the operations of Line 5. Lastly, since the DSP is not needed for the subtraction, we merge the subtraction step with the squaring and accumulation of $s_i^{(1)}$ coefficients. Once the last subtract-square-accumulate is complete, the total accumulation can be compared with bound $\lfloor \beta^2 \rfloor$.

IV. LEAF: The Proposed Accelerator

A. Overall Design Strategy

In keeping with our goal of lightweight resource usage, each component of the proposed LEAF accelerator is optimized for minimal resource usage. Generally, we follow the fetch-process-store strategy, where polynomial coefficients are fetched from the data memory, processed in a configurable arithmetic core, and stored back in the data memory again.

B. Component Descriptions

The top-level diagram of the proposed accelerator is shown in Fig. 2. The architecture can be broken down into seven components: (i) Decoding, (ii) Keccak Hashing Core, (iii) Data Memory, (iv) Twiddle Memory, (v) Polynomial Arithmetic Core, (vi) Norm Accumulator, and the (vii) Control Unit. Detailed component descriptions are given below.

Decode and Decompress. The Decode and Decomp modules implement the `Decode` and `Decomp` functions (Lines 3-4 in Algorithm 1), operating with bit-manipulation over a stream of incoming pk and sig bytes, producing $\boldsymbol{h}$ and $\boldsymbol{s^{(2)}}$, respectively. In general, both modules contain a buffer of the last 4 bytes received, and a small state machine that reads the newest incoming byte and produces 14-bit output words until all bytes of pk and sig have been processed. Since Decode and Decomp process 1 byte per cycle, each process takes as many cycles as the byte-size of pk and sig (see Table I).

Keccak Hashing Core. As mentioned, we utilize the "mid-range" Keccak core to minimize resource usage. There is a custom wrapper around the core to implement the `HashToPoint` function to hash $msg \,\|\, nonce$ to $\boldsymbol{c}$ (Line 2 of Algorithm 1). The core absorbs 8 bytes at a time until 136 bytes have been absorbed. Then we trigger the core to hash

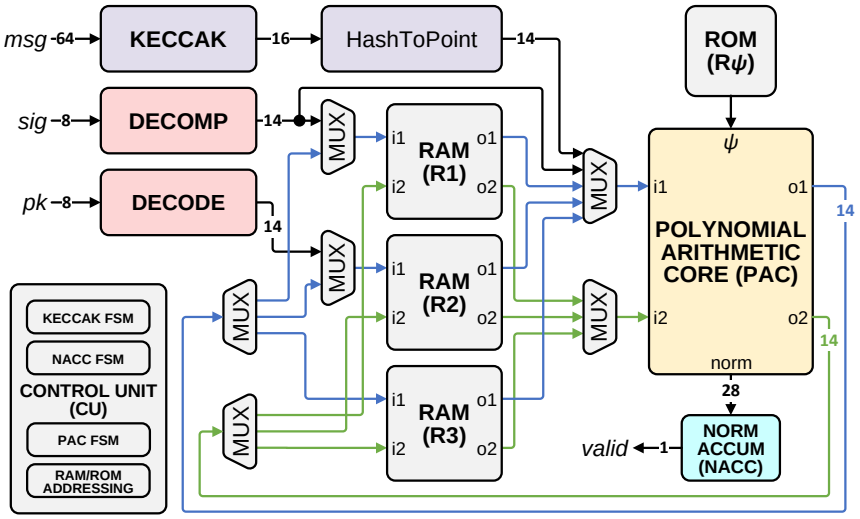


Fig. 2. Proposed LEAF: top-level diagram.

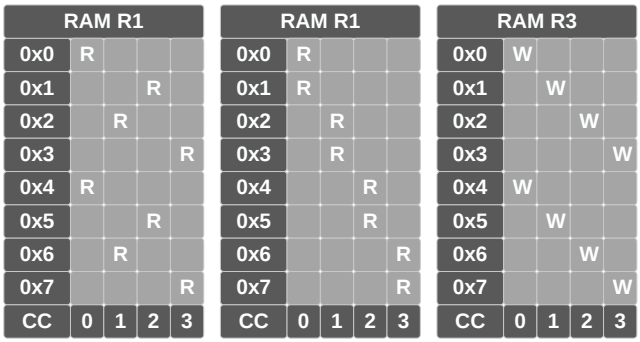


Fig. 3. Unified RAM access pattern for NTT/INTT. Toy example with $n = 8$. (a) Stage $= 1$ read pattern (bit-reverse order), (b) Stage > 1 read pattern, (c) Write pattern.

the ingested bytes, and this process repeats until all bytes of $msg \,\|\, nonce$ have been absorbed and hashed. If msg is less than 1.30kB or 2.85kB for $n = 512$ or $n = 1,024$ respectively, the wrapper waits for some time until $\boldsymbol{c}$ is needed. Then we trigger the core to squeeze, producing 8 bytes per cycle. We take the top 2 bytes as a $sample$, and discard if $sample$ is greater than 61,444 [5]. If $sample$ is within the 61,444 range, we pass along $sample \bmod q$. The process repeats until n valid $sample$'s have been produced for $\boldsymbol{c}$.

Data Memory (R1, R2, R3). Data Memory is responsible for storing and producing operands throughout the procedure, consisting of three true dual-port block RAMs ($R1, R2, R3$), each of size $n \times 14$. The access pattern for Data Memory during NTT/INTT operations is shown in Fig. 3. Initially, $R1$ and $R2$ are loaded with $\boldsymbol{h}$ and $\boldsymbol{s^{(2)}}$ values as they exit the Decode and Decomp blocks. Thereafter, operands ping-pong between $R1$, $R2$, and $R3$ to deliver and store values for the remaining steps of Algorithm 1.

Twiddle Memory (Rψ). Twiddle Memory is responsible for storing and producing twiddle factors ψ^i and ψ^{-i} for the NTT/INTT operations of Algorithm 1. This consists of a single-port ROM of size $n \times 14$. ψ^i values are pre-loaded in the ROM in order of when they need to be accessed for forward NTT. The storage and access pattern for the forward NTT ψ^i is shown in Fig. 4(a). By storing the values in order of when they are needed, forward NTT addressing can be implemented with simple counters in the Control Unit. The INTT address resolver is shown in Algorithm 4, with the access pattern illustrated in Fig. 4(b). We exploit the property of $\psi^{-(n-i)} = q - \psi^i$, so this pattern gives the address for the ψ^i value that, when subtracted from q in the PAC, produces the $\psi^{-(n-i)}$ required for the current INTT step.

Algorithm 4: ROM INTT Address Resolver

Input : n, $stage$, cc (From Algorithm 3)
Output: $R\psi$ $address$ (Fig. 4b)
Procedure
1 $r \leftarrow \log_2(n) - stage$
2 $hold \leftarrow 2^r(stage - 1) + (cc \bmod 2^r)$
3 $addr \leftarrow 2^r((1 \ll (stage - 1)) - 1) \,\|\, \mathrm{BR}(hold[r - 1 : 0])$
4 **Return** $n - addr - 1$
$\mathrm{BR}(\cdot)$: bit-reversal of number

Polynomial Arithmetic Core (PAC). The PAC implements all the arithmetic functions needed for Lines 5 and 6 of Algorithm 1. The PAC is shown in Fig. 5, along with a legend and a table of multiplexer configurations for various modes. The PAC involves just a single DSP. In the table in Fig. 5, the $s0 - s3$ values index the MUXs from top to bottom, and the ✓ indicates if an i/o port is in use for that PAC mode.

There are 5 configurable modes for the PAC: (a) Forward NTT Butterfly, (b) INTT Butterfly, (c) PWM, (d) Squaring, and (e) Subsquaring (modular subtraction followed by squaring). Modes (a) and (b) are used to implement the NTT/INTT Butterfly operations (Lines 8-9, 14-15 of Algorithm 3). Note that for INTT, the incoming ψ^i is subtracted from q to produce the required $\psi^{-(n-i)}$. Mode (c) is used for the PWM after NTT and before INTT (Line 5 ($\otimes$) of Algorithm 1) and implements a simple modular multiplication. Modes (d) and (e) are used for squaring (Line 6 of Algorithm 1). For both modes, the coefficients are normalized in ($\frac{-q}{2}$ to $\frac{q}{2}$)

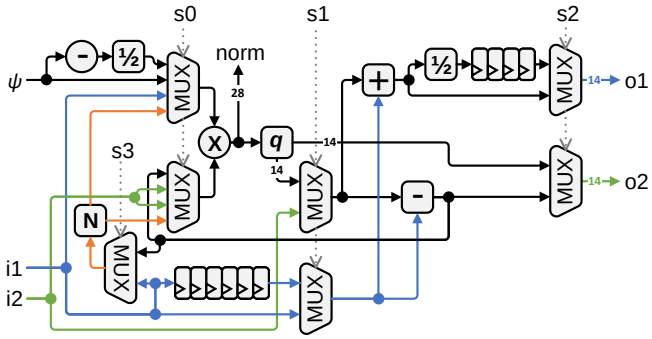
ROM Rψ													
ADDR	ψ^i	NTT Access Pattern											
0x1	ψ^4	R	R	R	R								
0x2	ψ^2					R	R						
0x3	ψ^6							R	R				
0x4	ψ^1									R			
0x5	ψ^3										R		
0x6	ψ^5											R	
0x7	ψ^7												R
CC		0	1	2	3	0	1	2	3	0	1	2	3
STAGE		1				2				3			

(a)

ROM Rψ															
ADDR	ψ^i	$q - \psi^i$	INTT Access Pattern												
0x1	ψ^4	ψ^{-4}									R	R	R	R	
0x2	ψ^2	ψ^{-6}							R		R				
0x3	ψ^6	ψ^{-2}						R		R					
0x4	ψ^1	ψ^{-7}					R								
0x5	ψ^3	ψ^{-5}			R										
0x6	ψ^5	ψ^{-3}				R									
0x7	ψ^7	ψ^{-1}	R												
CC			0	1	2	3	0	1	2	3	0	1	2	3	
STAGE			1				2				3				

(b)

Fig. 4. Twiddle factor ROM ψ access pattern. Toy example with $n = 8$. (a) NTT access pattern, (b) INTT access pattern.



N	Normalize in [-q/2 to q/2]	+	Modular addition
X	Normal multiplication (DSP)	-	Modular subtraction
-	Normal subtraction from q	½	Modular mul. by ½ mod q
	Register	q	Modular reduction

PAC MODE	s0	s1	s2	s3	ψ	i1	i2	o1	o2	norm
NTT	1	0	1	x	✓	✓	✓	✓	✓	x
INTT	0	1	0	x	✓	✓	✓	✓	✓	x
PWM	2	x	0	x	x	✓	✓	x	✓	x
SQUARE	3	x	x	1	x	✓	x	x	x	✓
SUBSQUARE	3	1	x	0	x	✓	✓	x	x	✓

Fig. 5. LEAF's Polynomial Arithmetic Core (PAC).

before squaring. Lastly, Subsquaring mode (e) performs the modular subtraction (Line 5 of Algorithm 1) before normalization/squaring.

Norm Accumulator (NACC). The NACC is used to perform the accumulation step for Line 6 of Algorithm 1 to compute the L2 norm of $\boldsymbol{s^{(1)}}$ and $\boldsymbol{s^{(2)}}$ and ensure they are shorter than $\lfloor \beta^2 \rfloor$. The component receives 28-bit squared values from the PAC and accumulates them in a 32-bit register. When the squares for all $\boldsymbol{s_i^{(1)}}$ and $\boldsymbol{s_i^{(2)}}$ have been accumulated, the total is compared with $\lfloor \beta^2 \rfloor$ and a *valid* signal is sent, ending the Signature Verification procedure.

C. Scheduling and Dataflow Example

This section details the operation scheduling throughout the accelerator during processing. Fig. 1 gives a high-level view of operation scheduling, as well as some direction on data movement between $R1$, $R2$, and $R3$. It is assumed that sm is parsed into sig, msg, and $nonce$ before processing (Line 1 of Algorithm 1). From there, the procedure is as follows:

On startup, the accelerator begins receiving bytes of $msg \| nonce$, pk, and sig. The Keccak core begins absorbing $msg \| nonce$, the Decode block processes pk, and Decomp processes sig, all in parallel. $\boldsymbol{s_i^{(2)}}$ and $\boldsymbol{h_i}$ coefficients populate $R1$ and $R2$ as they are produced. At the same time, $\boldsymbol{s_i^{(2)}}$ coefficients are also sent to the PAC where they are normalized, squared, and then accumulated in the NACC.

The Decomp block finishes producing $\boldsymbol{s^{(2)}}$ first. Since the blocks process one byte per cycle, this takes around 617 or 1,232 CCs for $n = 512$ and $n = 1,024$, respectively. At this point, all $\boldsymbol{s_i^{(2)}}$ are stored in $R1$, and we begin computing $\texttt{NTT}(\boldsymbol{s^{(2)}})$ with the PAC in Forward NTT Butterfly mode. During each NTT stage, $\boldsymbol{s_i^{(2)}}$ values under processing ping-pong between $R1$ and $R3$, while ψ^i are fed from $R\psi$. In Fig. 1, note that for $n = 1,024$, since 10 NTT stages are required, results end up back in $R1$, while for $n = 512$, 9 NTT stages are required, and so final results end up in $R3$.

The Decode block finishes producing $\boldsymbol{h}$ while $\texttt{NTT}(\boldsymbol{s_i^{(2)}})$ transform is being processing. With all $\boldsymbol{h_i}$ stored in $R2$, we begin computing $\texttt{NTT}(\boldsymbol{h})$. For $n = 512$, $\boldsymbol{h_i}$ values ping-pong between $R1$ and $R2$ (since $\boldsymbol{s^{(2)'}}$ is occupying $R3$) until results end up in $R2$. For $n = 1,024$, the values ping-pong between $R1$ and $R3$, ending back in $R1$.

At this point, $\boldsymbol{h'}$ and $\boldsymbol{s^{(2)'}}$ are in NTT form and can be multiplied point-wise. The PAC is switched to PWM mode, and we begin reading $\boldsymbol{h_i}$' and $\boldsymbol{s^{(2)}}$' values from $R1$ and $R3$ for $n = 512$, or $R1$ and $R2$ for $n = 1,024$. No value is needed from $R\psi$ since this is not an NTT operation. Results are stored in $R2$ for $n = 512$ or $R3$ for $n = 1,024$.

Once the PWM is finished, it is time to convert the resulting polynomial (located in $R2$ for $n = 512$ and $R3$ for $n = 1,024$) back using the INTT. The PAC is switched to INTT Butterfly mode. Thanks to the unified NTT/INTT of Algorithm 3, the read and write patterns from Fig. 3 are the same as for the forward NTT process. The difference is that now we use the INTT ROM access pattern from Fig. 4(b) (or see Algorithm 4 for how address is computed) to access ψ^i, which gets subtracted from q in the PAC to produce $\psi^{-(n-i)}$.

When the INTT transform completes, the resulting coefficients are stored in $R3$ for both $n = 512$ and $n = 1,024$. Now we check if all $msg \| nonce$ have been absorbed and hashed in the Keccak core, and wait if the process is not complete. If all $msg \| nonce$ is absorbed and hashed, we

TABLE II
COMPARISON WITH EXISTING NTT-BASED POLYNOMIAL MULTIPLICATION WORKS BASED ON FPGA IMPLEMENTATION

Device	Design	n	q	LUT/FF/Slice	DSP	BRAM	ENS[c]	Fmax (MHz)	CCs	Latency (μs)	ATP[d] (k)	eATP[e] (k)
Virtex-7	[21]	512	12,289	2,178/1,694/-[a]	8	3	2,088.4	234.0	1,074	4.6	10.00	9.59
Artix-7	[22]			741/330/-[a]	2	5	1,417.4	245.0	1,417[b]	5.8	4.29	8.20
	LEAF-NTT			898/567/305	1	1.5	701.7	251.3	2,394	9.5	8.6	6.7
Virtex-7	[21]	1,024	12,289	2,151/1,475/-[a]	9	2.5	2,084.3	234.0	2,104	9.0	19.3	18.7
	[25]			575/-/-[a]	3	11	2,645.1	125.0	5,160	41.3	23.7	109.2
Artix-7	[22]			847/375/-[a]	2	6	1,646.7	244.0	2,825[b]	11.6	9.8	19.1
	[23]			343/493/-[a]	3	6	1,591.6	251.0	6,206	24.7	8.48	39.35
	[24]			4,938/2,982/-[a]	8	0	2,362.3	153.0	2,560[b]	16.7	82.62	39.53
	LEAF-NTT			844/524/274	1	2	768.8	253.2	5,220	20.6	17.4	15.9

[a]: #Slice data not provided. We estimate as #Slice=#LUT/3.2, following the proposed design's #LUT/#Slice ratio (the bigger ratio case).
[b]: Cycles added for bit-reverse re-ordering operation. [c]: #DSP and #BRAM have been converted to ENS. [d]: ATP=#LUT×Latency. [e]: eATP=#ENS×Latency.

squeeze 2-byte $sample$'s from the Keccak core, and pass along ($sample \bmod q$) if $sample \leq 61,444$, to produce coefficients of $\boldsymbol{c_i}$. Note that this step stalls if $sample > 61,444$. There is a 93.8% chance that the $sample$ falls in the desired range, so [5] directs to oversample by 205 and 287 for $n = 512$ and $n = 1,024$, respectively, which results in 717 and 1,311 CCs to sample with a probability of $1 - 2^{-256}$ to produce enough $\boldsymbol{c_i}$. With the PAC set to Subsquaring mode, we read polynomial multiplication results from $R3$. The $\boldsymbol{c_i}$ values and coefficients from $R3$ are subtracted (to produce $\boldsymbol{s_i^{(1)}}$ coefficients), normalized, and squared in the PAC. The squared result is passed to the NACC to be accumulated.

Once all squared $\boldsymbol{s_i^{(1)}}$ coefficients have been accumulated in the NACC (with the previously accumulated $\boldsymbol{s_i^{(2)}}$), the result is compared with the bound $\lfloor \beta^2 \rfloor$, at which point it is determined if the signature is $valid$ or $invalid$, and the Signature Verification procedure is complete.

V. LEAF: EVALUATION AND COMPARISON

A. Performance Metrics Consideration

For a fair comparison, we use the aggregate resource utilization metric of Equivalent Number of Slices (ENS). We then use the Area-Time Product (ATP = $\#LUT \times Latency$), and Equivalent Area-Time Product (eATP = $\#ENS \times Latency$). While ATP only accounts for LUT usage, ENS is an aggregate measurement of Slice, DSP, and BRAM usage, so eATP is more balanced. Since some designs in Tables II and III did not report Slice utilization, so we estimate it according to the LUT/Slice ratios of our own designs, respectively, for a fair comparison. Note that the existing fixed LUT/Slice ratio estimation method like the one proposed in [38] is not always matching the actual implementation result (see LEAF's results in Tables II and III), so we just use the specific ratio from our design (the bigger ratio case) for all works in the same table. Finally, the converting ratios of #DSP and #BRAM to ENS are the same as [38] (which are also used in [7], [39] and others), i.e., 1 DSP equals 102.4 ENS, 1 BRAM(18K) equals 116.2 ENS, and 1 BRAM(36K) equals 196.2 ENS.

B. Experimental Setup

To test the effectiveness of the proposed LEAF, we have implemented the design for both $n = 512$ and $n = 1,024$ on the field-programmable gate array (FPGA) platform. The experimental setup is as follows: (a) The accelerator code is written in VHDL with functionality verified in ModelSim. (b) We have implemented the design in Vivado 2020.2 on the Artix-7 (xc7a200tfbg484-3) device. The FPGA utilization and frequency results are post place-and-route. (c) To showcase the efficiency of the proposed NTT core, we have also implemented the NTT-based polynomial multiplication and listed results in Table II. Note that only needed components are included, like NTT, PWM, and INTT, so components like Keccak core are not included.

C. LEAF-NTT (NTT-Based Polynomial Multiplication): Comparison and Analysis

The implementation results of the LEAF's NTT-based polynomial multiplication architecture are given in Table II. We have also listed some recent NTT-based polynomial multiplication architectures that operate over the same parameter set to FALCON, specifically the ones with lower ENS since that is our target. For $n = 512$, [22] is the most competitive with our design in terms of ENS and eATP, but our NTT still achieves a 50% reduction in ENS, as well as an 18% reduction in eATP. Notably, their design suffers from additional cycles for the bit-reversal operation, which our CG-based NTT/INTT does not require. It is worth noting that [21] is pipelined for consecutive NTT operations, so if the two forward NTT transforms are pipelined together, their design gains a further advantage in latency and achieves about the same eATP as ours. However, given that [21] was implemented on the superior Virtex-7 platform, and we still achieve a 66% reduction in ENS, we conclude that our design is superior for this application.

When $n = 1,024$, the analysis for [21], [22] is the same. The design in [23] opted for an out-of-place implementation for data memory despite using an in-place, GS-based NTT. They also simply pre-store all twiddle factors ψ for NTT and INTT instead of dynamically computing. Both of these decisions were likely made for simplicity, since this design is part of a larger ISA-based crypto-processor. To their credit, they do also implement NTT for 2 sets of twiddle factors, as a means to potentially support multiple PQC schemes. So for a fair comparison, we remove 1 BRAM from their usage. Still, our design achieves a 45% reduction in ENS and a 54% reduction in eATP. Regarding [24], like our design, they also used an out-of-place CG-based NTT, but with 4 butterfly cores. Since they use multiple cores, they spread their data memory to

TABLE III
COMPARISON WITH EXISTING FALCON SIGNATURE VERIFICATION WORKS BASED ON FPGA IMPLEMENTATION

Device	Design	n	LUT/FF/Slice	DSP	BRAM	ENS[b]	Fmax (MHz)	CCs	Latency (μs)	ATP[c] (K)	eATP[d] (K)
Ultrascale+	HLS'23 [11]	512	11,500/8,000/-[a]	15	13	7,469.0	-	-	618.0	7,107.0	4,615.8
	HW/SW'23 [14]		22,356/13,181/-[a]	13	54	18,501.3	-	314,639	-	-	-
Artix-7	HLS'20 [10]		57,604/17,764/-[a]	26	18	23,136.4	77.3	77,000	996.3	57,391.0	23,050.8
	HW'22 [8]		14,500/7,287/-[a]	4	2	5,066.7	142.0	2,399	16.9	245.0	85.6
	LEAF		4,104/2,621/1,203	1	1.5	1,697.8	236.4	9,036	38.2	156.9	64.9
Ultrascale+	HLS'23 [11]	1,024	13,300/8,600/-[a]	15	14	8,194.6	-	-	1,258.0	16,731.4	10,308.8
	HW/SW'23 [14]		22,356/13,181/[a]	13	54	18,501.3	-	613,911	-	-	-
Artix-7	HLS'20 [10]		58,645/18,241/-[a]	28	18	23,647.3	76.5	161,000	2,103.3	123,348.3	49,737.5
	HW'22 [8]		13,956/6,737/-[a]	4	2	4,906.7	142.0	4,687	33.0	460.6	162.0
	LEAF		3,991/2,571/1,218	1	2	1,712.8	237.0	19,235	81.2	324.0	139.0

[a]: Slice data not provided. We estimate as #Slice=#LUT/3.4 according to the #LUT/#Slice ratio of LEAF (the bigger ratio case).
[b]: #DSP and #BRAM have also been converted to ENS. [c]: ATP=#LUTs×Latency. [d]: eATP=#ENS×Latency. HW/SW: hardware-software co-design.

many RAM blocks, and implement them as distributed, which is why no BRAM is used. They use 2 DSPs per butterfly, compared to ours which uses just 1. Further, they also suffer from a long delay for the bit-reverse re-ordering step, which doubles their latency. With this, our design uses 67% and 60% less ENS and eATP, so we conclude our design is superior.

Lastly, we acknowledge that there exist some higher-radix-based NTT designs like [26]–[28] that do outperform our NTT architecture in eATP. This is expected, since our NTT is Radix-2, and increasing to Radix-4 or higher can provide a latency improvement at a rate disproportionate to the increase in resource usage. However, for our application, further accelerating the NTT computation would shorten the time available for the Keccak core to hash the incoming msg, and thus lower the size of msg that can be processed without delays. On the other hand, our design has the lowest ENS of any design for this parameter set (e.g., 41%, 65%, and 89% lower than [26], [27], and [28]), so we conclude that the proposed NTT design approach is a better choice for this application.

D. LEAF: Comparison and Analysis

The implementation results of the full LEAF on the FPGA platform are given in Table III along with the state-of-the-art hardware related designs for FALCON Signature Verification.

The designs from [10], [11] are based on the HLS design method and clearly suffer from significant inefficiencies in both area and timing. Our accelerator achieves between 97% and 99% reductions in eATP compared to both of these designs for both $n = 512$ and $n = 1,024$, as seen from Table III.

The design in [8] is the only known full-hardware accelerator for the Signature Verification procedure, and represents the state-of-the-art in the literature. They target high-performance implementation, so many of its design choices differ from ours. First, [8] uses a Radix-4 NTT with a 2×2 butterfly arrangement. Meanwhile, they employ the "high-speed" variant of the Keccak core, which allows them to process larger msg's without bottlenecks, but causes a significant increase in LUT/FF usage. Lastly, their norm accumulation unit is implemented separately from their arithmetic core without using DSPs, which also contributes to greater resource usage. Likely for simplicity, they implement their NTT for $n = 512$ by adding a conditional bypass, which is why their results for $n = 512$ actually use more resources than for $n = 1,024$. For $n = 512$, we achieve a 66% reduction in ENS, and a 24% reduction in eATP. For $n = 1,024$, LEAF uses 65% less ENS, and has 14% lower eATP. From this, we conclude that [8] certainly has some advantages for applications where processing very large msg's is common, but ours makes significant improvements for applications in resource-constrained environments. Most importantly, our proposed accelerator has a much better overall area-time complexity than [8].

Lastly, [14] uses the hardware/software co-design method based on RISC-V to implement the Signature Verification procedure. Since their architecture also includes support for the Dilithium PQC scheme, and meanwhile, it is not a full-hardware design, it is difficult to make a fair, direct comparison of resource usage. However, the latency (CCs) and area they provided are significantly longer than our design, so we conclude that our design is more efficient than [14].

E. Discussion and Future Work

Based on the above comparison and analysis, it is clear that both our NTT-based polynomial multiplication and full LEAF accelerator provide clear advantages over the existing works. Even comparing with the only existing full-hardware work of [8] targeting high-performance applications, our design still shows considerable strength in overall area-time complexities. Apart from the potential impacts to the crypto hardware design community, this work can also be a beneficial piece for the application design research community as our design is optimized for resource-constrained environments.

Next step would be to develop innovative techniques for other phases of FALCON and/or the other PQC schemes, as well as related design technique innovations such as side-channel resistant acceleration for FALCON.

VI. CONCLUSIONS

In this paper, we present a lightweight (first work in the field) and efficient accelerator for the Signature Verification procedure of the FALCON digital signature scheme. We have carefully selected hardware-friendly algorithmic primitives, implemented them with efficient hardware-level design techniques, formulated a highly efficient operation schedule to maximize utilization of the limited components we use, constructed a full accelerator (LEAF) on the FPGA platform, and demonstrated its efficiency compared with the existing designs. We hope this work can stimulate more work in the field.

ACKNOWLEDGEMENT

Part of this work was supported by NSF CNS-2453370. S. Coulon and J. Xie would like to thank Sports & Performance Grant of the College of Engineering at Villanova University.

REFERENCES

[1] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th annual symposium on foundations of computer science*, pp. 124–134, Ieee, 1994.

[2] D. J. Bernstein and T. Lange, "Post-quantum cryptography," *Nature*, vol. 549, no. 7671, pp. 188–194, 2017.

[3] PQC standardization process: Announcing four candidates to be standardized, plus fourth round candidates. https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4.

[4] J. Xie, W. Zhao, H. Lee, D. B. Roy, and X. Zhang, "Hardware circuits and systems design for post-quantum cryptography—a tutorial brief," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 71, no. 3, pp. 1670–1676, 2024.

[5] FALCON: Fast-Fourier Lattice-based Compact Signatures over NTRU. https://falcon-sign.info/.

[6] S. Mandal and D. Basu Roy, "Design of a lightweight fast fourier transformation for FALCON using hardware-software co-design," in *Proceedings of the Great Lakes Symposium on VLSI 2024*, pp. 228–232, 2024.

[7] S. Coulon, T. Bao, and J. Xie, "FELIX: FPGA-based scalable and lightweight accelerator for large integer extended GCD," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2024.

[8] L. Beckwith, D. T. Nguyen, and K. Gaj, "High-performance hardware implementation of lattice-based digital signatures," *Cryptology ePrint Archive*, 2022.

[9] D. Chawla and P. S. Mehra, "A roadmap from classical cryptography to post-quantum resistant cryptography for 5G-enabled IoT: Challenges, opportunities and solutions," *Internet of Things*, p. 100950, 2023.

[10] D. Soni, K. Basu, M. Nabeel, N. Aaraj, M. Manzano, and R. Karri, "Hardware architectures for post-quantum digital signature schemes," pp. 31–41, 2020.

[11] M. Schmid, D. Amiet, J. Wendler, P. Zbindon, and T. Wei, "Falcon takes off - a hardware implementation of the falcon signature scheme," *Cryptology ePrint Archive*, 2023.

[12] E. Karabulut and A. 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)*, pp. 90–100, IEEE, 2024.

[13] Y. Lee, J. Youn, K. Nam, H. H. Jung, M. Cho, J. Na, J.-Y. Park, S. Jeon, B. G. Kang, H. Oh, *et al.*, "An efficient hardware/software co-design for FALCON on low-end embedded systems," *IEEE Access*, 2024.

[14] P. Karl, J. Schupp, T. Fritzmann, and G. Sigl, "Post-quantum signatures on RISC-V with hardware acceleration," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 2, pp. 1–23, 2024.

[15] G. Alsuhli, H. Saleh, M. Al-Qutayri, B. Mohammad, and T. Stouraitis, "Area and power efficient FFT/IFFT processor for FALCON post-quantum cryptography," *IEEE Transactions on Emerging Topics in Computing*, 2024.

[16] P. He, B. Mongirdas, Ç. K. Koç, and J. Xie, "LAMP: Efficient implementation of lightweight accelerator for polynomial multiplication, from Falcon to RBLWE-ENC," in *Proceedings of the Great Lakes Symposium on VLSI 2024*, pp. 210–215, 2024.

[17] Y. Tu, S. Bai, J. Xiong, and J. Xie, "Novel schoolbook-originated polynomial multiplication accelerators for NTRU-based PQC," in *5th NIST PQC Standardization Conference*, pp. 1–13, 2024.

[18] S. Coulon, P. He, T. Bao, and J. Xie, "Efficient hardware RNS decomposition for post-quantum signature scheme FALCON," *Asilomar Conference on Signals, Systems, and Computers*, pp. 19–26, 2023.

[19] E. Karabulut and A. Aysu, "Masking FALCON's floating-point multiplication in hardware," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, no. 4, pp. 483–508, 2024.

[20] Y. Ouyang, Y. Zhu, W. Zhu, B. Yang, Z. Zhang, H. Wang, Q. Tao, M. Zhu, S. Wei, and L. Liu, "FalconSign: An efficient and high-throughput hardware architecture for Falcon signature generation," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2025, no. 1, pp. 203–226, 2025.

[21] Z. Ye, R. C. C. Cheung, and K. Huang, "PipeNTT: A pipelined number theoretic transform architecture," *IEEE Transactions on Circuits and Systems*, vol. 69, no. 10, 2022.

[22] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu, "Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2, pp. 49–72, 2020.

[23] A. Jati, N. Gupta, A. Chattopadhyay, and S. K. Sanadhya, "SPQ-Cop: Side-channel protected post-quantum cryptoprocessor." Cryptology ePrint Archive, Paper 2019/765, 2019.

[24] Y. Xing and S. Li, "An efficient implementation of the NewHope key exchange on FPGAs," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 3, pp. 866–878, 2020.

[25] A. C. Mert, E. Karabulut, E. Öztürk, E. Savaş, and A. Aysu, "An extensive study of flexible design methods for the number theoretic transform," *IEEE Transactions on Computers*, vol. 71, no. 11, pp. 2829–2843, 2022.

[26] Y. Zhao, X. Liu, Y. Hu, and H. Xiao, "Design of an efficient ntt/intt architecture with low-complex memory mapping scheme," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 71, no. 1, pp. 400–404, 2024.

[27] X. Chen, B. Yang, S. Yin, S. Wei, and L. Liu, "CFNTT: Scalable Radix-2/4 NTT multiplication architecture with an efficient conflict-free memory mapping scheme," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 94–126, 2022.

[28] S. Mandal and D. B. Roy, "Winograd for NTT: A case study on higher-radix and low-latency implementation of NTT for post quantum cryptography on FPGA," *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–14, 2024.

[29] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[30] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology — CRYPTO' 86* (A. M. Odlyzko, ed.), (Berlin, Heidelberg), pp. 311–323, Springer Berlin Heidelberg, 1987.

[31] Z. Cao, R. Wei, and X. Lin, "A fast modular reduction method," *Cryptology ePrint Archive*, 2014.

[32] E. Ozturk, Y. Doroz, E. Savas, and B. Sunar, "A custom accelerator for homomorphic encryption applications," *IEEE Transactions on Computers*, vol. 66, no. 1, 2017.

[33] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[34] W. M. Gentleman and G. Sande, "Fast fourier transforms-for fun and profit," 1966.

[35] M. C. Pease, "An adaptation of the fast fourier transform for parallel processing," *Journal of the ACM*, vol. 15, pp. 252–264, 1968.

[36] Y. Su, B.-L. Yang, C. Yang, and S.-Y. Zhao, "ReMCA: A reconfigurable multi-core architecture for full rns variant of BFV homomorphic evaluation," *IEEE Transactions on Circuits and Systems*, vol. 69, 2022.

[37] Keccak Team. https://keccak.team/.

[38] W. Liu, S. Fan, A. Khalid, C. Rafferty, and M. O'Neill, "Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 10, 2019.

[39] J. Ren, G. Du, Z. Li, X. Jia, Q. Liao, X. Wang, and D. Zhang, "An efficient ring polynomial multiplication accelerator for homomorphic encryption," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 71, no. 1, pp. 415–419, 2024.