Efficient Hardware RNS Decomposition for Post-Quantum Signature Scheme FALCON

Samuel Coulon, Pengzhou He, Tianyou Bao, and Jiafeng Xie
Department of Electrical and Computer Engineering, Villanova University, Villanova PA 19085 USA
Email: {scoulon,phe,tbao,jiafeng.xie}@villanova.edu

***Abstract*—The recently announced National Institute of Standards and Technology (NIST) Post-quantum cryptography (PQC) third-round standardization process has released its candidates to be standardized and FALCON is one of them. On the other hand, however, very few hardware implementation works for FALCON have been released due to its very complicated computation procedure and intensive complexity. With this background, in this paper, we propose an efficient hardware structure to implement residue numeral system (RNS) decomposition within NTRUSolve (a key arithmetic component for key generation of FALCON). In total, we have proposed three stages of coherent interdependent efforts to finish the proposed work. First, we have identified the necessary algorithmic operation related to RNS decomposition. Then, we have innovatively designed a hardware structure to realize these algorithms. Finally, field-programmable gate array (FPGA)-based implementation has been carried out to verify the superior performance of the proposed hardware structure. For instance, the proposed hardware design involves at least 3.91x faster operational time than the software implementation. To the authors' best knowledge, this is the first paper about the hardware acceleration of RNS decomposition for FALCON, and we hope the outcome of this work will facilitate the research in this area.**

***Index Terms*—Hardware design, FALCON, post-quantum cryptography, signature scheme, RNS decomposition.**

I. INTRODUCTION

It is known that the existing public-key cryptography (PKC) schemes like Rivest Shamir Adleman (RSA) and Elliptic Curve Cryptography (ECC) are vulnerable to the attacks launched from those well-established quantum computers [1]. Therefore, post-quantum cryptography (PQC) has drawn significant attention from various research communities recently [2]. Most importantly, the National Institute of Standards and Technology (NIST) PQC standardization process has just released its final selected algorithms to be standardized last July, including one public encryption scheme and three signature schemes [3].

Among these selected schemes, FALCON is considered quite unique [4], partly because of its complicated algorithmic procedure and partly due to its involved super-large computational complexity. Actually, since its initial submission, there have been very few implementation works released for FALCON. The software implementations can be seen in its original submission package [4] as well as a constant-timing one of [5]. While on the hardware implementation side, one high-level synthesis (HLS)-based result for FALCON verification (the simplest operation involved) was reported in [6] and then the same arithmetic component was also implemented in [7]. These works are the only representatives in the field.

Along with the PQC standardization process, more efforts have been gradually switching to the hardware implementation side [8]–[11]. Especially considering the fact that no complete hardware implementation work for FALCON has been released, it becomes ever more important to take the initiative to do the implementation work for FALCON. In this paper, we follow this direction to present a novel hardware-implemented residue numeral system (RNS) decomposition within NTRUSolve (the key arithmetic component for the key generation step of FALCON [4]), which is the first try in the field. Overall, we have carried out three layers of innovative works, including:

First of all, we have identified the key arithmetic operation related to the RNS decomposition of NTRUSolve (of FALCON), along with several critical algorithms that can successfully execute the targeted operation.

Then, we have proposed a novel hardware structure to accelerate the targeted operation within FALCON, based on novel algorithm-to-architecture mapping and optimization techniques.

Finally, we have conducted a thorough implementation-based comparison and analysis to showcase the efficiency of the proposed hardware RNS decomposition, e.g., it involves much less latency time than the related software implementation.

The rest of the paper is arranged as follows. Section II gives the targeted arithmetic operation and related algorithms to execute this operation. Section III gives a detailed description of the proposed RNS decomposition accelerator. Section IV presents a thorough implementation-based comparison, and Section V delivers the final conclusions.

II. ALGORITHM

This section introduces the targeted arithmetic operation within NTRUSolve of FALCON [4] and also the related algorithms to execute this operation.

Notations. These notations are used throughout the whole paper (interested readers may also refer to the original submission package of [4] for more details). n is the security level of FALCON; f, g, F, and G are polynomials of degree n with coefficients of 8 bits which solve the NTRU equation and make up the core private key of FALCON; and f', g', F', and G' are intermediate polynomials passed up or down as NTRUSolve is called recursively. In this paper, N, V, and U are the polynomial degree and coefficient size parameters, where V is the degree, U is the number of 31-bit elements making up each coefficient, and $N = V \times U$. Other notations in Algorithm 2 to Algorithm 5 can be seen later.

Brief Introduction of FALCON. FALCON is a lattice-based post-quantum signature scheme, which stands for **Fa**st Fourier **l**attice-based **co**mpact signatures over **N**TRU (N-th

degree truncated polynomial ring units). Overall, FALCON is the consummated product of many previous years' work, including: (i) the first signature scheme NTRUSIGN [12]; (ii) a generic framework to build hash-and-sign lattice-based signature schemes [13]; (iii) a provably secure NTRUSIGN was then proposed in [14]; (iv) an efficient implementation of identity-based encryption over NTRU lattices was proposed in [15]; (v) a new algorithm was then proposed in [16] to reduce the signing time. The current version of FALCON was invented by a group of experts in the field and then submitted to the NIST PQC standardization process and eventually was selected to be standardized [3].

NTRUSolve of FALCON. NTRUSolve is a critical arithmetic component for key generation of FALCON [4], as shown in Algorithm 1. Basically, NTRUSolve uses **Reduce** (see [4]) to reduce the size of F and G. The detailed mathematical and algorithmic principles underlying NTRUSolve can be seen in [4], [17]. For a successful hardware implementation of NTRUSolve, we observe the arithmetic operations involved within Lines 9-10 are critical as they determine how NTRUSolve will be carried out [4]. These two lines of operations (highlighted in Algorithm 1), though seem to be simple, they require several significant procedures to execute them, as discussed below.

Algorithm 1: FALCON NTRUSolve [4]

Require: $f, g \in \mathbb{Z}[x]/(x^n+1)$ with n a power of two;
Ensure : Polynomials F, G;

1 **if** $n = 1$ **then**
2 Compute $u, v \in \mathbb{Z}$ such that $uf - vg = \gcd(f, g)$
3 **if** $\gcd(f, g) \neq 1$ **then**
4 abort and return $\perp$
5 $(F, G) \leftarrow (vq, uq)$
6 **end**
7 return (F, G)
8 **else**
9 $f' \leftarrow N(f)$
10 $g' \leftarrow N(g)$
11 $(F', G') \leftarrow$ **NRTUSolve**$_{n/2,q}(f', q')$
12 $F \leftarrow F'(x^2)g(-x)$
13 $G \leftarrow G'(x^2)f(-x)$
14 **Reduce**(f, g, F, G)
15 **end**
16 return (F, G);

Overall Problem Statement. Based on the algorithmic sequence and software source code of FALCON in [4], it is seen that FALCON key generation requires we solve the NTRU equation, which is shown in Algorithm 1. One core operation of this algorithm is to find the field norm of coefficient inputs f and g, denoted f' and g' (Lines 9-10). FALCON does so by converting the integer coefficient inputs to RNS form modulo many prime values and then to RNS-NTT (number theoretical transform) form using the same primes. In RNS-NTT form, point-wise Montgomery Multiplication can be used to easily find f' and g'. Further, with each recursive call of NTRUSolve, the coefficients of f and g, and therefore also f' and g', grow large gradually and must be taken modulo more primes to be represented accurately. As such, the conversion from RNS-NTT back to RNS and further back to integer form is required with each recursive call of NTRUSolve. It is evident that an efficient RNS decomposition is an essential first step for implementing the full NTRUSolve algorithm, as well as FALCON as a whole.

Specific Algorithms. Following the above problem statement, we present here the needed Montgomery Multiplication (and others) for the targeted operation within NTRUSolve. Note that we describe this algorithm (as well as the following ones) based on the source code and original documentation of [4].

Algorithm 2: Montgomery Multiplication

Input : a, b, p, and $\overline{p}$ (a and b are the multiplier and multiplicand respectively, p and $\overline{p}$ for modular reduction, all 31-bit values ($\overline{p} = -1/p \bmod 2^{31}$);
Output: RES $= a \times b \bmod p$;

Main step
1 $z = a \times b$
2 $s = z[30..0] \times \overline{p}$
3 $w = s[30..0] \times p$
4 $d = w[61..0] + z[61..0]$
Final step
5 RES $= d[61..31] \bmod p$

Algorithm 2 describes the needed Montgomery Multiplication for Algorithm 1, which enables the multiplication of two 31-bit values with the result modulo a 31-bit value p without the need for expensive division operations. Algorithm 2 also calls for a 4th input $\overline{p}$, which is the inverse of p. Note that there occur a total of three multiplications, see Lines 1, 2, and 3, where each is implemented with a schoolbook multiplication (multiply, shift, and accumulate), and therefore each requires 31 cycles. With 2 final cycles for modular addition and subtraction, Algorithm 2 in total takes 95 cycles from input to output.

Algorithm 3: Normal Multiplication of Large and Small Integers

Input : x, y, and $\overline{F}_0$ (x is the multiplier, a large int sectioned into U-number 31-bit values, y is the multiplicand, a 31-bit value, and $\overline{F}_0$ is a 31-bit RNS value;
Output: z $= x \times y$;

Main step
1 $cc = 0$
2 **for** $i = 0$ *to* $U - 1$ **do**
3 $w = x[i] \times y$
4 $w = w + \overline{F}_0$
5 $w = w + cc$
6 $cc = w[61..31]$
7 $z[i] = w[30..0]$
8 **end**
Final step
9 z$[U] = cc$

Algorithm 3 implements a hardware-compatible schoolbook multiplication (multiply, shift, accumulate) of a large integer (x)

with a smaller, 31-bit integer (y). As will be explored in more detail later, all large values in FALCON are divided into 31-bit elements ($x[0], x[1]...x[n]$). This algorithm allows for those 31-bit elements to be taken sequentially, multiplied with the small integer, and accumulated with previous results to build a result equivalent to $x \times y$. On iteration $i = 0$, $x[i]$ is multiplied with y to produce a 62-bit result. Then, adding $\overline{F}_0$, which may be '0' in some use cases. After that, adding cc, which will be '0' on iteration 0. Finally, Algorithm 2 outputs the lower 31 bits to $z[i]$ and passes the higher 31 bits to cc for the next iteration. This process repeats until y has been multiplied with all elements of x that the final higher 31 bits cc are outputted to $z[n]$. z now contains $x \times y$, and z is 31 bits larger than the initial input x.

Algorithm 4: Converting from Integer to RNS

Input : F, p, $\overline{p}$, $R2$, and Rx (F is a polynomial of degree V with coefficients divided into U-number 31-bit elements, p, $\overline{p}$, $R2$, and Rx are arrays of U-number 31-bit values);
Output: $\overline{F}[j][i] = F[j] \bmod p[i]$;

Main step
1 **for** $i = 0$ *to* $U - 1$ **do**
2 **for** $j = 0$ *to* $V - 1$ **do**
3 $g = 0$;
4 **for** $z = U - 1$ *to* 0 **do**
5 $g = \text{Monty\_mul}(g,\ R2[i],\ p[i],\ \overline{p}[i])$ // see Algorithm 2
6 $w = F[j][z] \bmod p[i]$
7 $g = w + g \bmod p[i]$
8 **end**
9 **if** $F[j] < 0$ **then**
10 $g = g - Rx[i] \bmod p[i]$
11 **end**
12 $\overline{F}[j][i] = g$
13 **end**
14 **end**

Algorithm 4 describes the process to convert an integer polynomial F to an RNS polynomial $\overline{F}$ modulo primes $p[U-1..0]$. Polynomials F and $\overline{F}$ are both of degree V, with coefficients sectioned into U number of 31-bit elements. The first loop (Line 1) iterates through p values, beginning with $p[0]$. The second loop (Line 2) iterates through the coefficients of F, beginning with $F[0]$. The third loop (Line 4) iterates through the elements of the current F coefficient, beginning with $F[0][U-1]$ and decrements. On iteration 0, we take $F[0][U-1] \bmod p[0]$ and set it in g (Line 6). Note that Lines 5 and 7 yield no change since $g = 0$ is set as a start. With each subsequent iteration z, we execute the Montgomery multiplication with the previous result $g \times R2[0] \bmod p[0]$ and save the result back in g (Line 5). Then, we get the next element of the current F coefficient $F[0][z]$, set it as $\bmod p[0]$ (Line 6), accumulate it in the previous result g, and again $\bmod p[0]$ (Line 7). This process continues until we have iterated through all elements of the current coefficient $F[0]$. At this point, g contains $F[0] \bmod p[0]$. Lastly, if the initial $F[0]$ is a negative value, we subtract $g - Rx[0] \bmod p[0]$ from the result and save it back in g (lines 9-11). Eventually, the RNS value is complete and can be saved in $\overline{F}[0][0]$. The second loop (Line 2) continues to iterate until all coefficients of F have been reduced $\bmod p[0]$. The first loop (Line 1) continues to iterate until all coefficients of F have been reduced $\bmod p[U-1..0]$.

Algorithm 5: Converting from RNS to Integer

Input : $\overline{F}$, p, $\overline{p}$, $R2$, and s (F is a polynomial of degree V with coefficients divided into U-number 31-bit elements, p, $\overline{p}$, $R2$, and s are arrays of U-number 31-bit values);
Output: $F = rns\_to\_int(\overline{F})$;

Initial step
1 $tmp[0] = p[0]$
Main step
2 **for** $i = 1$ *to* $U - 1$ **do**
3 **for** $j = 0$ *to* $V - 1$ **do**
4 $xp = \overline{F}[j][i]$
5 $xq = \overline{F}[j][i-1..0] \bmod p[i]$ with $\overline{p}[i], R2[i]$ // see Algorithm 4
6 $xs = xp - xq \bmod p[i]$
7 $xr = \text{Monty\_mul}(s[i],\ xs,\ p[i], \overline{p}[i])$ // see Algorithm 2
8 $\overline{F}[j][i..0] =$ $\text{norm\_mul}(tmp[i-1..0], xr, \overline{F}[j][i-1..0])$ // see Algorithm 3
9 **end**
10 $tmp[i..0] = \text{norm\_mul}(tmp[i-1..0],\ p[i],\ 0)$
11 **end**
Final step
12 $F = \overline{F}$

Algorithm 5 depicts the process of converting an RNS polynomial $\overline{F}$ to an integer polynomial F. Polynomials $\overline{F}$ and F are both of degree V, with coefficients decomposed into U number of 31-bit elements. The overall computation process is supported by an array of 31-bit values tmp, which represents all previous p values multiplied together (i.e. during iteration $i = 3$ of the outer loop, $tmp = p[0] \times p[1] \times p[2]$). In the beginning, we set $p[0]$ in $tmp[0]$ (Line 1). The first loop (Line 2) iterates through p values, beginning with $p[1]$. The second loop (Line 3) iterates through the coefficients of $\overline{F}$, beginning with $F[0]$.

On iteration 0, the algorithm takes the first element of $\overline{F}[0]$ and saves it in xp (Line 4), and then reduces to $\overline{F}[j][0] \bmod p[1]$ using $\overline{p}[1]$ and $R2[1]$ (following the process described in Algorithm 4, but skipping Lines 9-11 of Algorithm 4 because $\overline{F}$ values are unsigned). Next, the algorithm subtracts $xp - xs \bmod p[1]$ and saves the result in xs (Line 6). Furthermore, the algorithm multiplies $s[1] \times xs$ using Montgomery Multiplication and saves the result in xr (Line 7). Lastly, the algorithm multiplies all values of tmp, which is just $p[0]$ at this point, with xr, while also accumulating with existing value $\overline{F}[j][1]$ (Line 8). This process continues until we have iterated through all coefficients of $\overline{F}$. Before the next iteration of the

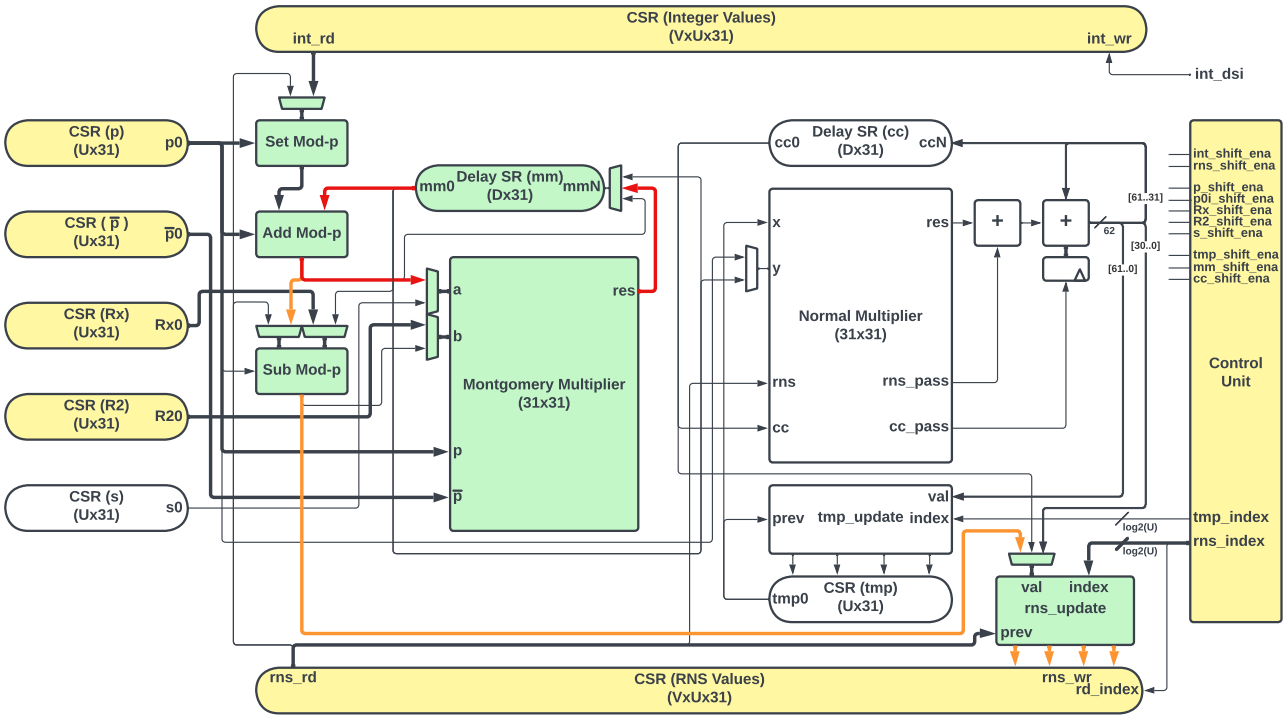

Fig. 1. The proposed architecture for RNS decomposition, during integer to RNS conversion (highlighted components and data-flow lines denote that they are used during this process). CSR: circular shift-register.

first loop, the algorithm multiplies $tmp \times p[1]$ to yield the next tmp value.

For subsequent iterations of the first loop (Line 2), it is important to note that the time to complete each iteration grows exponentially with i. The time for operations on Lines 5, 7, 8, and 10 all grow as i increases. For example, for the operation on Line 5, on iteration 0, we need only find $\overline{F}[j][1] \bmod p[1]$, which requires a single iteration of Algorithm 4. However, on iteration i, we need to find $\overline{F}[j][i] \bmod p[1]$ for all $\overline{F}[j][i..0]$, which requires i iterations of Algorithm 4.

The above algorithms (Algorithm 2 to Algorithm 5) contain the necessary operations to execute the targeted RNS decomposition in Algorithm 1. The following section will present the corresponding architecture mapped from these algorithms.

III. Proposed Hardware Architecture

This section presents the proposed hardware-implemented RNS decomposition for Falcon (NTRUSolve).

Overall Architecture. The overall architecture of the proposed hardware design is shown in Fig. 1, 2, and 3 (various stages of processing). The overall architecture consists of one control unit, five circular shift registers (CSRs) for precomputed values p, $\overline{p}$, Rx, $R2$, and s, one CSR tmp for holding accumulated p values paired with one component tmp_update which indexes and loads tmp values, three components for modular reduction ($set \bmod p$), addition ($add \bmod p$), and subtraction ($sub \bmod p$), one CSR holding initial integer F values, one CSR holding resulting RNS $\overline{F}$ values paired with a component rns_update which indexes and loads RNS values, one 31×31 Montgomery Multiplier, one 31×31 Normal Multiplier, and two delay shift registers holding the results mm and cc from the multiplication components.

Functions of Components. The control unit is responsible for generating enable signals for various sub-components of the system. The control unit consists of a top-level state machine as well as sub-level state machines which control data flow throughout the system.

The p, $\overline{p}$, Rx, $R2$, and s CSRs (left side of Fig. 1, 2, and 3) hold pre-computed values that are required by Algorithms 3, 4, and 5, where each contains U 31-bit elements. Note that the sizes of these shift registers match the sizes of F and $\overline{F}$ coefficient values. During processing, these registers shift such that the index of the element at their output ($p0$, $\overline{p}0$, etc.) matches the index of the F or $\overline{F}$ element being processed (i.e., $p[i]$, $\overline{p}[i]$, $Rx[i]$, $R2[i]$, $s[i]$ is available for processing $F/\overline{F}[i]$).

The tmp CSR (blue in Fig. 2) is responsible for feeding tmp values to the Normal Multiplication unit during the later phase of RNS to integer conversion (Algorithm 5, Line 8). The tmp_update component is responsible for updating the tmp shift register with new values, and placing them in a specific index depending on the phase of the process (Algorithm 5, Line 10). The current values of the tmp shift register are passed to the Normal Multiplication unit and multiplied with the current p value to yield the new tmp value, which is then saved back in the tmp shift register.

The $set \bmod p$, $add \bmod p$, and $sub \bmod p$ components are used to perform modular arithmetic. Component $set \bmod p$ simply takes a 31-bit element and subtracts p if the input is greater than p (Algorithm 4, Lines 6). Component $add \bmod p$ adds two 31-bit elements and subtracts p if the result is greater than p (Algorithm 4, Lines 7). Component $sub \bmod p$ subtracts two 31-bit elements and subtracts p if the result is greater than p (Algorithm 4, Line 10).

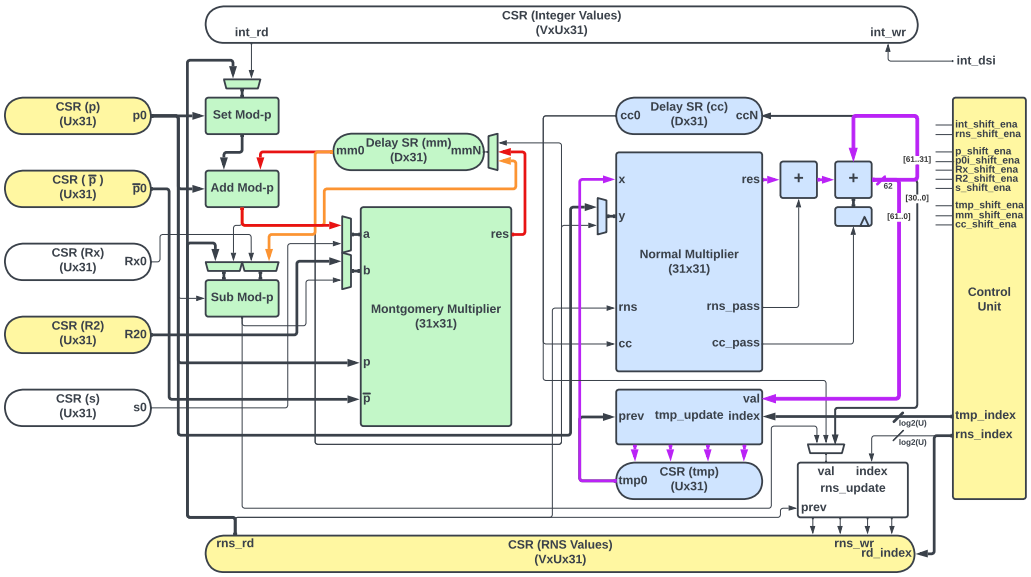

Fig. 2. The proposed architecture for RNS decomposition, during the first stage of RNS to integer conversion.

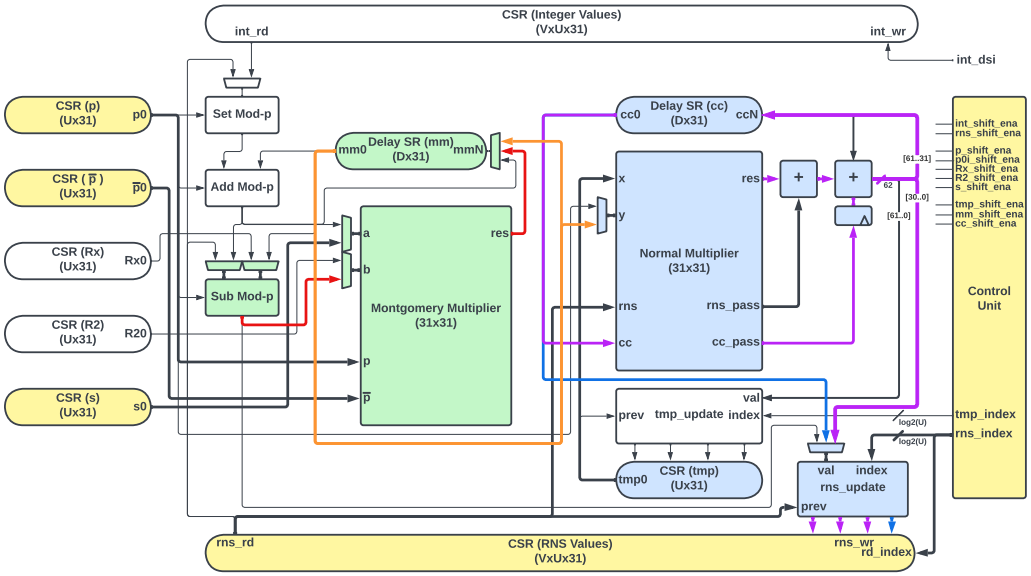

Fig. 3. The proposed architecture for RNS decomposition, during the second stage of RNS to integer conversion.

The F data shift registers (Fig. 4) is responsible for feeding F integer coefficient values to the system during the integer to RNS conversion. First, during the data-shift-in state, 31-bit elements are sequentially shifted in for N cycles until all N initial coefficient elements are loaded in respective registers. From this point on, F values are handled in coefficient groups, where the N 31-bit elements are now sectioned into V groups of U 31-bit elements. Coefficients are indexed as $F[V]$ and individual coefficient elements are indexed as $F[V][U]$, where the first element shifted-in is loaded in $F[0][0]$ (coefficient 0, element 0). Per Algorithm 4, the read port of this shift register is linked to register $U-1$, which is $F[0][U-1]$ to start. During processing, coefficient groups are shifted downward through the shift register, where $F[i][j]$ moves to $F[i-1][j]$ on each cycle. Values of $F[0]$ are shifted around to the other end of the CSR. As this coefficient is shifted around, the elements within the coefficient are also shifted downward, where $F[0][U-1]$ moves to $F[V-1][0]$ and remaining elements $F[0][i]$ move to $F[V-1][i+1]$ (see Fig. 4). This dual shifting process provides the necessary effect to supply the correct F element to the system for executing Algorithm 4.

The $\overline{F}$ data shift register (Fig. 5) is responsible for saving results from the integer to RNS conversion, feeding $\overline{F}$ RNS values to the system (during the RNS to integer conversion), and saving results from the RNS to integer conversion (when RNS to integer conversion is done in place). $\overline{F}$ values are organized in the same way as F values, with coefficients indexed as $F[V]$ and individual coefficient elements indexed as $F[V][U]$. During integer to RNS, results are received sequentially in order of the p value used for reduction, so the first V results will all be reduced by $p[0]$.

The Montgomery Multiplication component (Fig. 6) per-

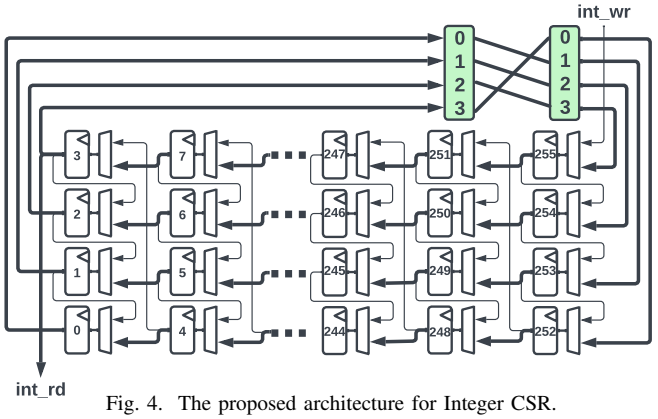

Fig. 4. The proposed architecture for Integer CSR.

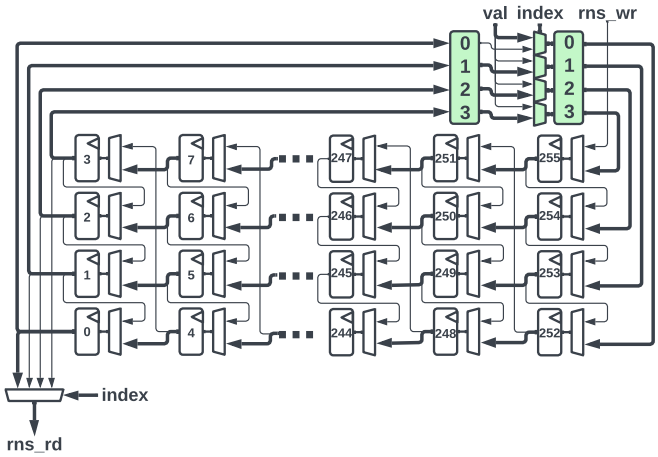

Fig. 5. The proposed architecture for RNS CSR and RNS update.

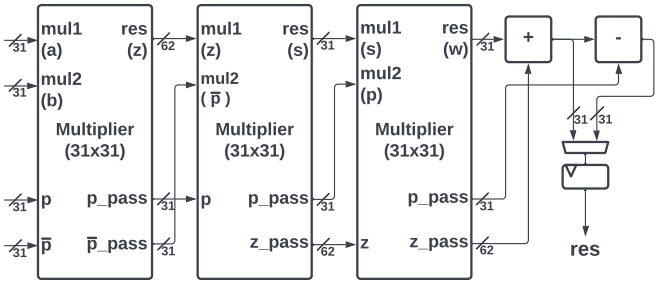

Fig. 6. The proposed architecture for Montgomery Multiplication.

forms a multiplication of two 31-bit values $\bmod p$. This component has been implemented as Algorithm 2, which employs three phases of schoolbook multiplication (multiply, shift, accumulate) and takes 95 cycles from input to output. For efficiency and ease of implementation, this component is designed to be fully pipelined, so new input values a, b, p, and $\overline{p}$ can be provided on every cycle and will propagate through the component to produce a result. The full pipelining of values p and $\overline{p}$ come at the expense of employing many registers whose only purpose is to pass values to a later phase of the multiplier (p_pass, $\overline{p}\_pass$, z_pass in Fig. 6). After multiplication, this component performs an optional subtraction for modular reduction and passes the result out. This component is paired with a delay shift register mm, which saves current multiplication results to be accumulated with later results per Algorithms 4 and 5.

The Normal Multiplication component (blue in Figs. 2 and 3) performs a schoolbook multiplication of two 31-bit values and produces a 62-bit result. The component is paired with two adders and a delay shift register cc to form a multiplication unit capable of multiplying a large integer with a small integer per Algorithm 3. At the completion of each 31×31 multiplication phase, the 62-bit result is divided into its higher and lower 31 bits. The lower 31 bits are saved, and the high 31 bits are passed around as the carry bits for the next 31×31 multiplication phase. This process is repeated until all 31-bit elements of the initial large integer have been multiplied with the 31-bit small integer, at which point the final 62-bit result is saved with previous results.

Data Flow – Integer to RNS. The process for converting integer F to RNS $\overline{F}$ is shown in Fig. 1, which realizes the operation of Algorithm 4. The process is divided into two stages: (1) a multiplication loop (red lines in Fig. 1) and (2) saving results (orange lines in Fig. 1).

Stage (1) implements Lines 3-8 of Algorithm 4. The F CSR begins shifting and outputting element $U-1$ of each F coefficient. These elements are passed through $set \bmod p$ with $p[0]$, passed to $add \bmod p$ with previous result mm (All mm are '0's on iteration 0) and $p[0]$, and passed to the Montgomery Multiplier to be multiplied with $R2[0]$, with $p[0]$ and $\overline{p}[0]$. Results of the Montgomery Multiplication are sequentially passed to delay shift register mm. Once $F[V-1][U-1]$ has been passed to $set{-}modp$, coefficient $F[0]$ arrives back at its start (but now shifted such that $F[0][U-2]$ is at the output). This element is passed to $set \bmod p$ with $p[0]$ and then to $add \bmod p$ where it is added with the previous result of $F[0][U-1]$ (which should now be exiting the mm delay register). The red lines in Fig. 1 show the data path for this multiply-accumulate process. This process is repeated until all elements U of all coefficients V have been passed through and reduced with $p[0]$.

Stage (2) executes Lines 9-11 of Algorithm 4. At this point, the fully accumulated results should be sequentially exiting in the mm delay shift register. These values are passed to $sub \bmod p$ where, if the original coefficient from which they were generated is negative, $Rx[0]$ is subtracted from them. Results are now in RNS and are sequentially sent to rns_update, where they are loaded into their respective $\overline{F}$ coefficients. For this iteration, the values are saved in $\overline{F}[V][0]$ since, as should be noted, they were all reduced by $p[0]$.

Stages (1) and (2) repeat U times until all elements U of all coefficients V have been reduced by all p values. At this point, $\overline{F}$ registers should be occupied by RNS values, where $\overline{F}[i][j] = F[j] \bmod p[i]$.

Data Flow – RNS to Integer. The process for converting RNS $\overline{F}$ to integer F is shown in Figs. 2 and 3 (Algorithm 5). This process is divided into four stages: (1) a multiplication loop (red and orange lines in Fig. 2); (2) updating tmp registers (purple lines in Fig. 2); (3) Montgomery Multiplication (red and orange lines in Fig. 3); and (4) Normal Multiplication (purple and blue lines in Fig. 3). Note that stages (1) and (2) occur in parallel and are represented together in Fig. 2. So these stages occur sequentially as: (1) and (2), (3), (4).

TABLE I
IMPLEMENTATION RESULTS OF RNS DECOMPOSITION ON THE INTEL STRATIX-V FPGA

degree	ALM	Fre.	Latency (Int. to RNS)[1]	Latency (RNS to Int.)[1]	Delay (Int. to RNS)	Delay (RNS to Int)
Polynomials of High Degree $V = 64$, Small Coefficient $U = 4$						
$V = 64$	15,879	243.55	1,392	1,374	5.715	5.642
Polynomials of Medium Degree $V = 16$, Medium Coefficient $U = 14$						
$V = 16$	15,241	247.46	17,360	17,636	70.153	71.268
Polynomials of Low Degree $V = 4$, Large Coefficient $U = 53$						
$V = 4$	18,423	227.48	259,382	264,689	1140.241	1163.570

Unit for Fre. (frequency): MHz. Delay=(1/Fre.)×latency, unit for Delay: μs.
[1]: Latency is the major computation cycles for a given processing phase.

Stage (1) realizes Lines 5-6 of Algorithm 5, which essentially calls Algorithm 4 (with the exception of Lines 9-11, because $\overline{F}$ values are unsigned), and thus the process is very similar to that of the integer to RNS conversion described above. To begin, we perform the process from Algorithm 4 with some key differences. In Line 2 of Algorithm 5, the iteration begins with $i = 1$, and in Line 3, we reduce elements of the current coefficient less than i. So, for iteration 0 of this stage, we perform Algorithm 4 with $p[1]$, $\overline{p}[1]$, and $R2[1]$ on only $\overline{F}[j][0]$. For subsequent iterations i we perform Algorithm 4 with $p[i]$, $\overline{p}[i]$, and $R2[i]$ for all $\overline{F}[j][i-1..0]$. When all $\overline{F}[j][i-1..0]$ have been reduced with $p[i]$, the resulting values xq are now in mm and are passed to $sub \bmod p$ component, where they are subtracted from $F[j][i]$ with $p[i]$ to produce xs. At this point, Stage (1) is completed, and values are ready to be passed to Stage (3).

Stage (2) executes Lines 1 and 10 of Algorithm 5 and occurs in parallel with Stage (1). While we carry out Stage (1), the Normal Multiplier is used to multiply the existing large tmp value with the current p value to determine the next tmp value (that is needed for multiplication in Stage (4)). On iteration 0, we simply set $p[0]$ in $tmp[0]$ per Line 1 in Algorithm 5. On iteration 1, the lone value in tmp (which is $p[0]$) is multiplied with the new p value, $p[1]$. The 62-bit result is separated into 31-bit elements. The lower 31 bits are saved back in $tmp[0]$. Since there is nothing else to accumulate with, the higher 31-bit carry value is saved back in $tmp[1]$ (tmp now contains $p[0] \times p[1]$). This process continues with each iteration i, and tmp always contains $p[0] \times p[1] \times ...p[i]$.

Stage (3) implements Line 7 of Algorithm 5. When Stages (1) and (2) are completed, xs values, the reduced $\overline{F}[j]$ values coming out $sub \bmod p$ component, are sent to the Montgomery Multiplier and multiplied with $s[i]$ using $p[i]$ and $\overline{p}[i]$. The results of the multiplication xr are passed to mm for the preparation of Stage (4).

Stage (4) is the execution of Line 8 of Algorithm 5. This final step is to multiply the 31-bit results from Stage (3) with the large value saved in tmp from Stage (2). On iteration 0, as previously stated, tmp contains only $p[0]$, so it is a simple 31×31 single iteration xr and $tmp[0]$ with the Normal Multiplier, and addition with the original $\overline{F[j][0]}$. This yields a 62-bit result, where the lower 31 bits are saved in $\overline{F}[j][0]$ and the higher 31 bits are saved in $\overline{F}[j][1]$. For subsequent iterations, tmp is larger and thus requires i iterations with the Normal Multiplier. On each iteration i, we multiply xr with the next 31-bit element of tmp. Since xr values need to be multiplied with all $tmp[i-1..0]$, xr values are sent back to mm to be delayed until they are needed for the next tmp element. At the output of the multiplier, the 62-bit result is added with $\overline{F}[j][i]$ as well as cc the carry 31-bits from the previous multiplication. The lower 31 bits are saved in $\overline{F}[j][i-1]$, and the higher 31 bits are passed back around through cc. This process repeats until xr has been multiplied with all 31-bit elements of tmp. The final 62-bit result is saved in $\overline{F}[j][U-1..U-2]$.

Stages (1), (2), (3), and (4) repeat $U - 1$ times until all V coefficients have been processed. At this point, $\overline{F}$ register should now be occupied by integer values. Again, note that now RNS to integer conversion has occurred in place. An optional final step would be to shift the resultant integer values in $\overline{F}$ back to F.

IV. IMPLEMENTATION & COMPARISON

We have coded the proposed hardware architecture (general architecture of Fig. 1) in VHDL (functionality verified through ModelSim, through the checking with FALCON reference implementation in [4]) and implemented it on Vivado 2021.2 (AMD-Xilinx UltraScale+ XCZU9EG-2FFVB1156 device) and Quartus 19.7 (Intel Stratix-V 5SGXMA9N1F45C2).

The related implementation results, such as the number of LUTs, FFs, Slices, adaptive logic modules (ALMs), and maximum frequency (Fre.), are obtained and listed in Tables 1 and 2. In a practical implementation of the targetted NTRUSolve function and RNS decomposition by extension, the degree of the initial polynomial V would be 1,024, and the size of its coefficients U would be 1. With each recursive call of NTRUSolve, V decreases by half, and U approximately doubles. As such, the dimensions V and U of the polynomial at each recursive call are known, and these predetermined V and U pairs were used during implementation (see the details in Tables I and II).

Discussion. It was found that the implemented hardware architecture performs favorably when the degree of the processing polynomial V is larger than the size of its coefficients U. This is somewhat expected as the initial idea behind the architecture targeted NTRUSolve of recursive depth as 4, where $V = 64$ and $U = 4$. Meanwhile, the pipelining of the proposed design does not adjust for recursive depths when U is much larger than V, which resulted in considerable dead processing time.

To demonstrate the efficiency of the proposed design, We have also measured the performance of the software imple-

TABLE II
IMPLEMENTATION RESULTS OF RNS DECOMPOSITION ON THE AMD-XILINX ULTRASCALE+ FPGA

degree	LUT	FF	Slice	Fre.	Latency (Int. to RNS)[1]	Latency (RNS to Int.)[1]	Delay (Int. to RNS)	Delay (RNS to Int.)
Polynomials of High Degree $V = 64$, Small Coefficient $U = 4$								
$V = 64$	15,169	26,789	2,437	371.75	1,392	1,374	3.744	3.696
Polynomials of Medium Degree $V = 16$, Medium Coefficient $U = 14$								
$V = 16$	15,276	25,281	2,620	416.67	17,360	17,636	41.664	42.3264
Polynomials of Low Degree $V = 4$, Large Coefficient $U = 53$								
$V = 4$	20,171	27,093	3,192	387.60	259,382	264,689	669.206	682.898

Unit for Fre. (frequency): MHz. Delay=(1/Fre.)×latency, unit for Delay: μs.
[1]: Latency is the major computation cycles for a given processing phase.

TABLE III
SOFTWARE IMPLEMENTATION RESULTS OF RNS DECOMPOSITION

Degree	Small Coefficient	Delay (Int. to RNS)	Delay (RNS to Int.)
$V = 64$	$U = 4$	22,363	15,306
$V = 16$	$U = 14$	66,985	44,750
$V = 4$	$U = 53$	240,398	161,649

Unit for Delay: ns.

mentation for RNS decomposition. The experimental setup is as follows: (i) we have used the microbenchmark support library from Google [18] as the benchmark library; (ii) we have used an AMD Ryzen Threadripper 3960X processor running at 3.8 GHz; (iii) the testing was carried out on the Ubuntu 20.04 LTS OS on a KVM-based virtual machine; (iv) we have used g++ 9.4.0 to compile the code and the benchmark running at a single-thread. Due to the data flow of the software implementation, the RNS to Int. (integer) transfer requires the data from the Int. to RNS function. The result of RNS to Int. is the time difference of Int. to RNS to Int. and Int. to RNS. The software implementations of Int. to RNS and RNS to Int. take 22,363/66,985/44,750 ns (number of testing iterations is 31,266/10,468/2,912) and 15,306/44,750/161,649 ns (number of testing iterations is 18,559/6,257/1,743) for $V = 64, U = 4$, $V = 16, U = 14$, and $V = 4, U = 53$ respectively.

It is clear that the proposed hardware implementation has better processing timing than the software one, e.g., for the transferring of integer (Int.) to RNS, the proposed hardware design (preferred parameter selection of $V = 64$ and $U = 4$) has at least 3.91x and 5.97x than the software implementation, respectively, on the Intel and AMD FPGA platforms.

V. CONCLUSION

This paper, for the first time, presents a novel hardware implementation of RNS decomposition for FALCON. We firstly presented the necessary algorithms that are essential to realize the targeted arithmetic operation within FALCON NTRUSolve. Then, we have presented the designed hardware architecture in a detailed format. Finally, we have implemented the proposed hardware architecture on the FPGA platform and have compared it with the software implementations to showcase the efficiency of the proposed design. It is expected the proposed work can initiate further work in the FALCON implementation as well as the ongoing NIST standardization process.

ACKNOWLEDGMENT

The work of J. Xie was supported by NIST-60NANB20D203 and NSF SaTC-2020625. S. Coulon was supported by the REU supplement fund of NSF SaTC-2020625.

This paper is accepted in 57th Asilomar Conference on Signals, Systems, and Computers.

REFERENCES

[1] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *35th sym. found. comp. scie.*, pp. 124–134, 1994.
[2] *Post-quantum cryptography*, https://csrc.nist.gov/projects/post-quantum-cryptography.
[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] T. Prest, P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon," *Post-Quantum Cryptography Project of NIST*, 2022.
[5] T. Pornin, "New efficient, constant-time implementations of falcon," *Cryptology ePrint Archive*, 2019.
[6] D. Soni and et al., *Hardware Architectures for Post-Quantum Digital Signature Schemes*. Springer, 2021.
[7] L. Beckwith, D. T. Nguyen, and K. Gaj, "High-performance hardware implementation of lattice-based digital signatures," *Cryptology ePrint Archive*, 2022.
[8] J. Xie and et al., "Special session: The recent advance in hardware implementation of post-quantum cryptography," in *VTS*, 2020.
[9] Y. Tu, P. He, Ç. K. Koç, and J. Xie, "Leap: Lightweight and efficient accelerator for sparse polynomial multiplication of hqc," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2023.
[10] B. J. Lucas and et al., "Lightweight hardware implementation of binary ring-lwe pqc accelerator," *IEEE Computer Architecture Letters*, vol. 21, no. 1, pp. 17–20, 2022.
[11] W. Tan and et al., "High-speed vlsi architectures for modular polynomial multiplication via fast filtering and applications to lattice-based cryptography," *IEEE Transactions on Computers*, 2023.
[12] J. Hoffstein, N. Howgrave-Graham, J. Pipher, J. H. Silverman, and W. Whyte, "Ntrusign: Digital signatures using the ntru lattice," in *Topics in Cryptology—CT-RSA 2003: The Cryptographers' Track at the RSA Conference 2003 San Francisco, CA, USA, April 13–17, 2003 Proceedings*, pp. 122–140, Springer, 2003.
[13] C. Gentry, C. Peikert, and V. Vaikuntanathan, "Trapdoors for hard lattices and new cryptographic constructions," in *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pp. 197–206, 2008.
[14] D. Stehlé and R. Steinfeld, "Making ntru as secure as worst-case problems over ideal lattices," in *Advances in Cryptology–EUROCRYPT 2011: 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings 30*, pp. 27–47, Springer, 2011.
[15] L. Ducas, V. Lyubashevsky, and T. Prest, "Efficient identity-based encryption over ntru lattices," in *Advances in Cryptology–ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, ROC, December 7-11, 2014, Proceedings, Part II 20*, pp. 22–41, Springer, 2014.
[16] L. Ducas and T. Prest, "Fast fourier orthogonalization," in *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, pp. 191–198, 2016.
[17] T. Pornin and T. Prest, "More efficient algorithms for the ntru key generation using the field norm," in *22nd IACR International Conference on Practice and Theory of Public-Key Cryptography*, pp. 504–533, Springer, 2019.
[18] "benchmark: A microbenchmark support library google.." https://google.github.io/benchmark/random_interleaving.html.