Post-Quantum HAWK Signature Acceleration with RISC-V-Based Hardware-Software Co-Design

Rishabh Shrivastava and Utsav Banerjee
Electronic Systems Engineering, Indian Institute of Science, Bengaluru, India
Email: rishabhs@iisc.ac.in, utsav@iisc.ac.in

***Abstract*—Advances in quantum computing technology have motivated the development of post-quantum cryptography (PQC) algorithms. HAWK is a new post-quantum digital signature scheme and the only lattice-based candidate selected for Round 3 of the "Additional Digital Signatures" phase of the NIST PQC Standardization process. HAWK offers compact key and signature sizes compared to NIST standard ML-DSA (Dilithium), and its simple design avoids the use of floating-point arithmetic unlike NIST standard FN-DSA (FALCON). This makes HAWK very well suited for resource-constrained applications. We perform software runtime profiling of HAWK signature computation and verification on a resource-efficient Vex RISC-V processor core, and identify Keccak permutations and polynomial transformations as the most computationally expensive functions. In this work, we demonstrate light-weight hardware-software co-design of HAWK with these operations accelerated using Vex RISC-V Custom Function Units and accompanying custom instructions. We present multiple design variants with different degrees of acceleration, and our best design achieves $\approx 3\times$ speedup and $\approx 40\%$ reduction in area-time-product compared to the baseline when implemented on a Xilinx Artix-7 FPGA.**

***Index Terms*—post-quantum cryptography, digital signature, NIST PQC, HAWK, lattice, Keccak, NTT, FFT, Vex RISC-V, CFU, custom instructions, hardware acceleration, FPGA, IoT.**

I. Introduction

Modern public key cryptography is based on the computational intractability of hard mathematical problems such as integer factorization and discrete logarithm. Shor's algorithm [1], if realized on a large-scale quantum computer, can solve these problems in polynomial time and break the security of these cryptographic schemes. Recent advances in the research and development of quantum computing technologies have motivated the theoretical design and practical implementation of post-quantum cryptography (PQC), that is, cryptographic algorithms which are secure against future quantum adversaries. The U.S. National Institute of Standards and Technology (NIST) has been driving the standardization of post-quantum cryptography (PQC) since 2016 [2]–[6]. After several years of theoretical analysis, security evaluation and optimized implementation, lattice-based digital signature schemes Dilithium [7] and FALCON [8] were recently selected by NIST as PQC standards. Since 2022, NIST has started an "Additional Digital Signatures" phase of the PQC Standardization process [9], [10], and HAWK [11], [12] is the only lattice-based scheme selected among its Round 3 digital signature candidates.

Lattice-based PQC schemes offer strong security guarantees against both classical and quantum adversaries. However, they have large memory overheads and involve complex polynomial arithmetic computations. This makes them particularly challenging to implement on resource-constrained embedded systems for applications such as the Internet of Things (IoT). HAWK offers compact public key and signature sizes, along with fast signature computation and verification without requiring floating-point arithmetic. This makes HAWK a suitable candidate for resource-constrained embedded systems for applications such as the Internet of Things (IoT). Discrete Gaussian sampling and polynomial transformations account for majority of the computation cost of the HAWK signature algorithms. Their core functions with strong data dependencies which become computational bottlenecks on general-purpose processor systems are Keccak State Permutation (KSP), Number Theoretic Transform (NTT) (with its inverse iNTT) and Fast Fourier Transform (FFT) (with its inverse iFFT).

We propose a light-weight hardware-software co-design of HAWK using the resource-efficient Vex RISC-V core [13]. We design a Custom Function Unit (CFU) [14], [15] to accelerate Keccak permutations and polynomial transformations using custom instructions which we incorporate into the standard RISC-V RV32IM Instruction Set Architecture [16]. We present multiple variants of our CFU design with varying degrees of acceleration: (I) Keccak only, (II) Keccak + NTT, (III) Keccak + NTT + FFT, and (IV) Keccak + (i)NTT + (i)FFT. Our best implementation (with CFU design variant II) achieves $\approx 3\times$ speedup compared to the baseline software implementation of HAWK signature computation and verification on the Vex RISC-V core, while also providing $\approx 40\%$ reduction in area-time-product (ATP) on Xilinx Artix-7 FPGA. Our implementation has very low FPGA hardware resource utilization, thus making it appropriate for constrained IoT applications.

II. Background

A. HAWK Post-Quantum Digital Signature Scheme

HAWK is a quantum-safe digital signature scheme based on the intractability of the Module Lattice Isomorphism Problem [11], intended to be secure against both classical and quantum adversaries. It is the only lattice-based scheme selected for

© 2026 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

This is an extended / revised version of the following paper with the same title published in the proceedings of the 2026 International Conference on VLSI Design (VLSID) - DOI: 10.1109/VLSID68508.2026.00088

Round 3 of the "Additional Digital Signatures" phase of the NIST PQC Standardization process. HAWK has smaller public key and signature sizes compared to the recent lattice-based NIST PQC standard ML-DSA based on Dilithium [7]. HAWK is also faster compared to the upcoming lattice-based NIST PQC standard FN-DSA based on FALCON [8]. HAWK is particularly well-suited for resource-constrained applications (such as Internet of Things) because of its simple design and high speed. Unlike FALCON, it does not require the use of floating point arithmetic, thus improving computational efficiency as well as side-channel security [12].

The HAWK specification recommends two main parameter sets HAWK-512 and HAWK-1024 at the NIST security levels 1 and 5 respectively. They involve computations with polynomials of degree 512 and 1024 respectively. The sizes of private key, public key and signature (in bytes) are 184, 1024 and 555 respectively for HAWK-512 and 360, 2440 and 1221 respectively for HAWK-1024. HAWK signature generation uses SHAKE256 [17] for hash computations and discrete Gaussian sampling, and the Number Theoretic Transform (NTT) and its inverse (iNTT) for efficient polynomial multiplications [18]. HAWK signature verification uses SHAKE256 for hash computations, NTT and NTT-adjoint for matrix inner product and matrix norm computations, and the fixed-point Fast Fourier Transform (FFT) and its inverse (iFFT) [19] for efficient polynomial division during signature decompression. Further details about the HAWK algorithms, parameter sets and security analyses are available in [11], [12].

B. SHAKE256

SHAKE256 is an extendable output function specified in the SHA-3 standard [17] which can produce outputs of arbitrary length. Like all members of the SHA-3 family, SHAKE256 is a sponge construction based on the Keccak-f[1600] permutation, which absorbs input bits into its internal state and squeezes output bits from the state. Keccak-f[1600] operates on a 1600-bit state which is internally represented as a $5 \times 5 \times 64$-bit array and permuted over 24 rounds per block of 1088 input / output bits. Each permutation round consists of five basic steps θ, ρ, π, χ and ι which can be implemented using bit-wise AND, NOT, XOR and rotation operations. Further details about SHAKE256 and the Keccak permutation are available in the SHA-3 standard [17].

C. NTT, iNTT and NTT-adjoint

Given a polynomial $a(x) \in \mathbb{Z}_q[x]/(x^N+1)$ with coefficients $a(x) = (a_0, a_1, \cdots, a_{N-1})$, its NTT representation $\hat{a}(x) = (\hat{a}_0, \hat{a}_1, \cdots, \hat{a}_{N-1})$ is defined as $\hat{a}_k = \sum_{m=0}^{N-1} a_m \zeta_N^{mk} \bmod q$ for $k \in \{0, 1, \cdots, N-1\}$. The polynomial $a(x)$ can be recovered from $\hat{a}(x)$ by performing the inverse operation (iNTT) as $a_m = \frac{1}{N} \sum_{k=0}^{N-1} \hat{a}_k \zeta_N^{-mk} \bmod q$ for $m \in \{0, 1, \cdots, N-1\}$. Here, ζ_N is the N-th primitive root of unity in the ring $\mathbb{Z}_q$, that is, $\zeta_N^N = 1 \bmod q$ and $\zeta_N^k \neq 1 \bmod q$ for all $k \neq N$. Its powers ζ_N^{mk} are known as twiddle factors. To support negative-wrapped convolution for polynomial multiplication [19], the modulus must also satisfy $q \equiv 1 \bmod 2N$ so that both the N-th and $2N$-th primitive roots of unity modulo q exist. The NTT representation of a self-adjoint polynomial [12], which we refer to as NTT-adjoint, can be computed by encoding only half of its coefficients, and is therefore faster.

D. FFT and iFFT

For an N-length complex sequence $x = \{x_0, x_1, \cdots, x_{N-1}\}$, its FFT is given by another N-length complex sequence $X = \{X_0, X_1, \cdots, X_{N-1}\}$ defined as $X_k = \sum_{m=0}^{N-1} x_m \omega_N^{mk}$ for $k \in \{0, 1, \cdots, N-1\}$. The sequence x can be recovered from X by performing the inverse operation (iFFT) as $x_m = \frac{1}{N} \sum_{k=0}^{N-1} X_k \omega_N^{-mk}$ for $m \in \{0, 1, \cdots, N-1\}$. Here, $\omega_N = exp(-2\pi j/N)$ is the N-th complex root of unity ($j = \sqrt{-1}$), and its powers ω_N^{mk} are known as twiddle factors.

E. Related Work

Since HAWK is a new digital signature scheme, and its selection for Round 3 of NIST PQC Additional Digital Signatures was announced very recently, there are very few implementations in current literature. Preliminary assembly-optimized software implementations for ARM Cortex-M4 were presented by [20], and these results were further improved by [12]. A RISC-V-based hardware-software co-design of HAWK was presented by [21] with custom SIMD (Single Instruction Multiple Data) instructions to accelerate Keccak, modular arithmetic, NTT and FFT. While the focus of [21] was high performance at the cost of logic overhead, our aim is to demonstrate a light-weight implementation suitable for resource-constrained IoT applications. We present design space exploration of HAWK with a light-weight RISC-V core integrating hardware accelerators for various important functions such as Keccak, (i)NTT and (i)FFT.

III. Hardware Architecture

A. Vex RISC-V and Custom Function Unit

Our design is based on the light-weight Vex RISC-V 32-bit micro-processor [13] which implements a 5-stage in-order pipeline supporting the RV32IM instruction set [16]. In

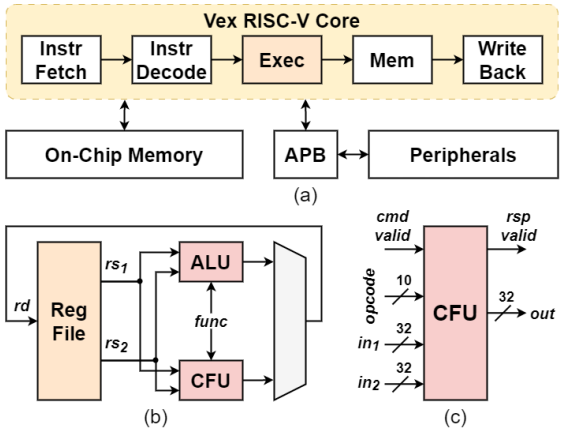


Fig. 1. (a) Murax SoC with 5-stage pipelined Vex RISC-V processor core, (b) Execute stage with ALU and CFU, and (c) CFU interface.

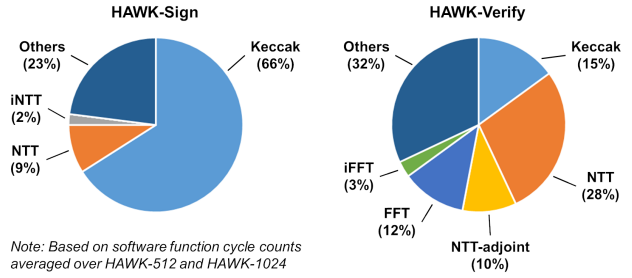


Fig. 2. Distribution of computation cost of HAWK among important functions based on reference software runtime profiling on Vex RISC-V.

particular, we use the Murax SoC (System-on-Chip) which integrates the Vex RISC-V core with shared instruction and data memory, Advanced Peripheral Bus (APB), GPIOs and UART for external interfacing. We use the Custom Function Unit (CFU) methodology [14], [15] to implement accelerators tightly coupled with the execution stage of the RISC-V processor pipeline to support new custom functions complementing the standard arithmetic and logic instructions. Figure 1 shows the high-level architecture of Vex RISC-V integrated with CFU. The 5 pipeline stages are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write Back (WB). The EX stage consists of the arithmetic logic unit (ALU) multiplexed with the CFU. They together interface with the register file with source addresses rs_1 and rs_2, and destination address rd. The CFU has 32-bit data inputs / outputs and works with a 10-bit opcode.

B. HAWK Software Runtime Profiling

We first perform detailed software profiling on the baseline Vex RISC-V processor using the isochronous reference implementation of HAWK [12]. The distribution of computation cost among the expensive functions – Keccak, (i)NTT, NTT-adjoint, (i)FFT and other functions is shown in Figure 2. In this work, we consider implementations of only the HAWK-Sign and HAWK-Verify algorithms which are used most frequently for signature computation and signature verification respectively. In comparison, HAWK-KeyGen, which is used for private and public keypair generation, is performed infrequently, especially in the context of IoT applications. We observe that SHA-3 Keccak permutations and polynomial transformations together account for ≈77% and ≈68% of HAWK-Sign and HAWK-Verify respectively, thus making them the most important targets for hardware-software co-design. Therefore, we accelerate HAWK with CFUs for computing the Keccak permutation, (i)NTT and (i)FFT. We propose different design versions with different levels of CFU integration.

C. Proposed Hardware-Software Co-Design Architecture

For our proposed hardware-software co-design of HAWK with Vex RISC-V and CFU, we use R-type RISC-V custom 32-bit instructions of the format `opcode dout, addr, din`. All operands are 32 bits wide to be compatible with the RISC-V RV32IM Instruction Set Architecture (ISA) [16], and

TABLE I
PROPOSED RISC-V CFU CUSTOM INSTRUCTIONS FOR HAWK

Name	Description
`writentt addr, din`	Write (i)NTT inputs into CFU Mem
`readntt dout, addr`	Read (i)NTT outputs from CFU Mem
`writefft addr, din`	Write (i)FFT inputs into CFU Mem
`readfft dout, addr`	Read (i)FFT outputs from CFU Mem
`writetf addr, din`	Write twiddle factors into CFU Mem
`params addr, din`	Write constants for (i)NTT / (i)FFT into CFU registers (transform size / prime / prime inverse)
`startntt`	Start NTT operation in CFU
`startintt`	Start iNTT operation in CFU
`startnttadj`	Start NTT-adjoint operation in CFU
`startfft`	Start FFT operation in CFU
`startifft`	Start iFFT operation in CFU
`writeksp addr, din`	Write Keccak inputs into CFU Mem
`readksp dout, addr`	Read Keccak outputs from CFU Mem
`startksp`	Start Keccak operation in CFU
`busybit dout, addr`	Read availability status from CFU register

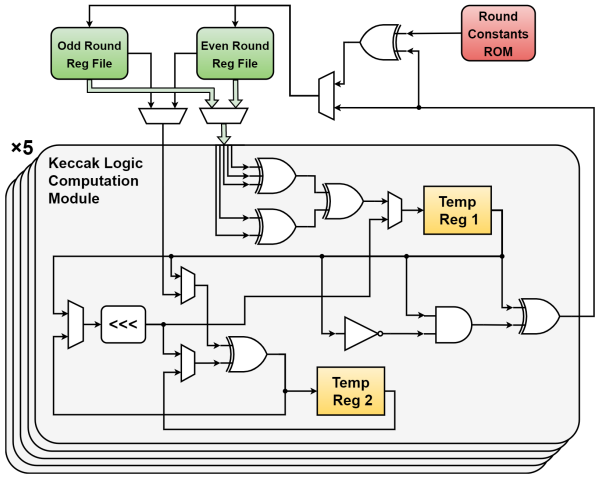


Fig. 3. Architecture of our proposed Keccak State Permutation Unit with a read-only memory for storing round constants, two register files for inputs / outputs, and five 64-bit logic computation modules.

the 10-bit `opcode` is obtained by concatenating the `f7` and `f3` fields of the 32-bit instruction. The proposed instructions are listed in Table I, and can be categorized into three groups - writing data from the processor to CFU memories or registers, starting execution of one of the supported functions, polling a status bit to check completion of execution, and reading data from CFU memories or registers to the processor. The `dout` field is unused for instructions writing to CFU, while the `din` field is unused for instructions reading from CFU. Instructions for function execution do not require any of the `dout`, `din` and `addr` fields since only the `opcode` is sufficient.

D. Keccak State Permutation Unit

Figure 3 shows the architecture of our Keccak State Permutation unit. It consists of a read-only memory to store round constants, a data memory bank to store the permutation inputs and outputs, registers for intermediate values, control circuitry and 5 logic computation modules which perform the constituent AND, NOT, XOR and rotation operations.

Algorithm 1 Keccak-f[1600] State Permutation Round with Plane-per-Plane Processing (adopted from [22])

Require: Input array of 64-bit lanes $A_{i,j} \, \forall i, j \in \{0, \cdots, 4\}$
Ensure: Output array of 64-bit lanes $E_{i,j} \, \forall i, j \in \{0, \cdots, 4\}$

for $i = 0; \, i < 5; \, i = i + 1$ **do** ▷ θ step
  $C_i \leftarrow A_{i,0} \oplus A_{i,1} \oplus A_{i,2} \oplus A_{i,3} \oplus A_{i,4}$
end for
for $i = 0; \, i < 5; \, i = i + 1$ **do** ▷ θ step
  $D_i \leftarrow C_{i-1} \oplus (C_{i+1} \lll 1)$
end for
for $j = 0; \, j < 5; \, j = j + 1$ **do**
  for $i = 0; \, i < 5; \, i = i + 1$ **do** ▷ θ, ρ and π steps
    $B_i \leftarrow (A_{i',j'} \oplus D_{i'}) \lll r_{i',j'}$ where $r_{i',j'}$ denotes the bit-wise rotation offset and $(i', j')^T = M^{-1}(i, j)^T$ with M being the permutation matrix for π step
  end for
  for $i = 0; \, i < 5; \, i = i + 1$ **do** ▷ χ step
    $E_{i,j} \leftarrow B_i \oplus ((\neg B_{i+1}) \wedge B_{i+2})$
  end for
end for
$E_{0,0} \leftarrow E_{0,0} \oplus RC_k$ where RC_k denotes the round constant for ι step in the k-th round ▷ ι step
return $E_{i,j} \, \forall i, j \in \{0, \cdots, 4\}$

The Keccak-f[1600] permutation requires 24 round constants, each of 64-bit size, which are stored in a 24 × 64-bit read-only memory. The input to the Keccak permutation is a 5 × 5 × 64-bit state array, which is written to the CFU memory using a series of `writeksp` instructions. The operation is then started using the `startksp` instruction. After the operation is completed, the permuted state output is read from the CFU memory using a series of `readksp` instructions. Since the operands in the CFU instructions are only 32-bit wide, each read / write operation for a 64-bit Keccak lane is performed using two instructions with appropriate addressing.

To achieve balance between performance and resource utilization, we implement *plane-per-plane processing* [22] which simultaneously operates on five 64-bit lanes of the state (together called a plane) using 5 Keccak logic computation modules. The high-level computation flow per round is shown in Algorithm 1. Each logic module operates on 64-bit data and contains a rotate-left circuit, a NOT gate, an AND gate and multiple XOR gates. We follow a multi-cycle architecture, allowing re-use of logic blocks across 6 clock cycles per round while preserving critical path timing. We have temporary registers to store intermediate results from computations in different cycles in a Keccak round. The CFU memory is implemented as a pair of 25 × 64-bit register files to accommodate any number of read / write operations per cycle. Once the state permutation operation is started, the two register files follow a ping-pong data flow between even and odd rounds. Each Keccak round takes 6 clock cycles, thus requiring total 144 clock cycles to complete the Keccak state permutation.

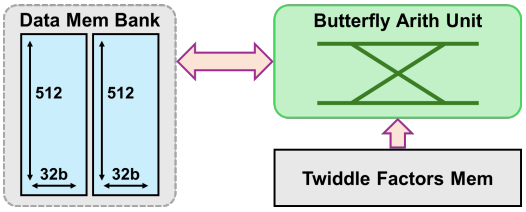


Fig. 4. Architecture of our proposed polynomial transformation unit with the butterfly arithmetic unit supporting either only NTT or unified NTT + FFT.

E. Polynomial Transformation Unit

Figure 4 shows our polynomial transformation unit which consists of a butterfly arithmetic unit, a dual-port data memory bank, a twiddle factor memory and control circuitry implementing an in-place data flow for (i)NTT and (i)FFT [23]. Different versions of our design support different levels of functional unification in the polynomial transformation unit. Therefore, the butterfly arithmetic unit is implemented as either a dedicated NTT butterfly, a unified NTT + FFT butterfly, or a unified (i)NTT + (i)FFT butterfly in different design versions. Our design supports polynomials with configurable degree to be compatible with both HAWK-512 and HAWK-1024 at different NIST security levels. All inputs, outputs and twiddle factors are 32-bit wide. The (i)NTT operands are unsigned integers in a prime field, while the (i)FFT operands are signed fixed-point complex numbers. All negative quantities are stored in standard two's complement format.

Computation of HAWK NTT butterfly requires one 32-bit multiplier, two 32-bit adders / subtractors and appropriate modular reduction circuitry. Simple conditional subtraction / addition is used to reduce the addition / subtraction outputs. A Montgomery reduction unit containing two 32-bit multipliers, one 64-bit adder, one 32-bit subtractor and a 32-bit conditional adder is used to reduce the multiplication output. The same arithmetic blocks are re-used for computation of HAWK iNTT butterfly apart from an extra conditional adder. Computation of HAWK FFT butterfly requires four 32-bit multipliers, two's complement units for signed multiplication inputs / outputs and six 64-bit adders / subtractors. The arithmetic blocks used in FFT and NTT butterflies are re-used for computation of HAWK iFFT butterfly. Our unified (i)NTT + (i)FFT butterfly unit demonstrates extensive resource sharing between different operating modes, as shown in the detailed circuit diagrams in Figure 5. The butterfly unit is implemented with a 6-stage internal pipeline for improved performance.

Our unified butterfly unit can compute either one (i)NTT butterfly or one (i)FFT butterfly at a time. HAWK-512 requires 512-point (i)NTT, 512-point NTT-adjoint and 256-point (i)FFT. HAWK-1024 requires 1024-point (i)NTT, 1024-point NTT-adjoint and 512-point (i)FFT. Our unified butterfly can be configured to accelerate all these polynomial transformations. Data inputs / outputs for 512-point (i)FFT are 512 complex numbers stored as 1024 32-bit elements in the data memory bank, and 768 complex twiddle factors are stored as 1536 32-bit elements in the twiddle factor memory. Data inputs /

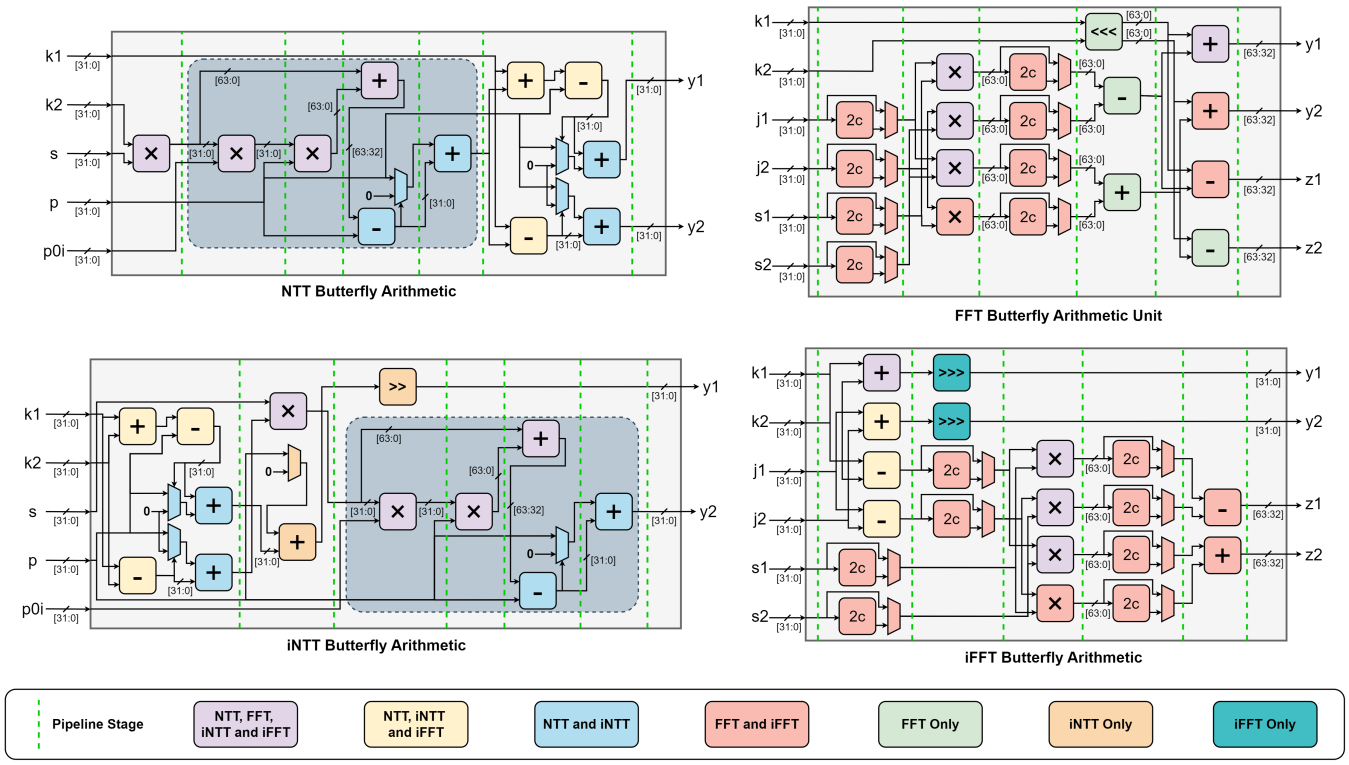


Fig. 5. Detailed architecture of our proposed butterfly circuitry for NTT, iNTT, FFT and iFFT. Sharing of various arithmetic circuits (adders, subtractors and multipliers) across different operating modes is shown with colour coding. Pipeline stages are shown as green dashed lines. Montgomery modular reduction circuits used in NTT and iNTT modes are shown in gray boxes. All bit-wise shift logic are implemented with simple wiring.

outputs for 1024-point (i)NTT are 1024 integers stored as 1024 32-bit elements in the data memory bank. Data inputs / outputs for 1024-point NTT-adjoint are 512 integers stored as 512 32-bit elements in the data memory bank. The twiddle factors for 1024-point (i)NTT and NTT-adjoint are 1024 integers stored as 1024 32-bit elements in the twiddle factor memory. The requirements of 256-point (i)FFT, 512-point (i)NTT and 512-point NTT-adjoint are similarly implemented using a subset of the above memory organization.

The data memory bank is implemented as a pair of 512 $\times$ 32-bit true-dual-port memories. The twiddle factor memory is implemented as a 1536 $\times$ 32-bit simple-dual-port memory. In the design version where we support only NTT and iNTT, twiddle factor memory is implemented as a 1024 $\times$ 32-bit simple-dual-port memory for resource savings. Before starting an (i)NTT or (i)FFT computation, the twiddle factor memory is configured through the CPU-CFU interface. In any given cycle during computation, 4 read / write transactions are performed on the data memory and a maximum of 2 read transactions are performed on the twiddle factor memory.

IV. Implementation Results

Our proposed hardware-software co-design of HAWK is implemented on a Digilent Arty evaluation board with a Xilinx Artix-7-100T FPGA device. We use the reference C software implementation provided with [12] and replace the computationally expensive Keccak permutation and polynomial transformation functions with our CFU-accelerated versions. Verilog Hardware Description Language is used to design the hardware accelerator, and Xilinx Vivado Design Suite version ML 2022.2 is utilized for FPGA synthesis, implementation and simulation. We implement multiple variants of the CFU integrated with Vex RISC-V processor to study the functional unification trade-offs: (I) Keccak State Permutation (KSP) only, (II) KSP + NTT, (III) KSP + NTT + FFT, and (IV) KSP + NTT + FFT + iNTT + iFFT. Table II compares the resource utilization of these design variants along with the baseline Vex RISC-V core without CFU integration. Table III compares the performance of these variants when executing HAWK-512 and HAWK-1024 signature computation and verification. As expected, higher degree of CFU integration leads to reduced cycle count at the cost of increased resource utilization and

TABLE II
Resource Utilization Summary of Design Variants on Artix-7

Design	Supported Functions	FPGA Resource Utilization LUT / FF / DSP / BRAM / Slice	Equiv. Slice †	Freq. (MHz)
(0)	Baseline (No CFU)	1935 / 1830 / 4 / 27 / 689	7373.4	100
(I)	KSP only	10271 / 7085 / 4 / 27 / 3516	10200.4	100
(II)	KSP + NTT	11722 / 6255 / 15 / 30 / 3777	12285.0	99
(III)	KSP + NTT + FFT	13386 / 7231 / 20 / 31 / 4348	13600.4	74
(IV)	KSP + (i)NTT + (i)FFT	15961 / 7744 / 20 / 31 / 4870	14122.4	68

† For Xilinx 7-series FPGAs, 1 DSP $\approx$ 102.4 slices and 1 BRAM $\approx$ 232.4 slices
$\Rightarrow$ Equivalent Slice Utilization $\approx$ Slice + (102.4 $\times$ DSP) + (232.4 $\times$ BRAM) [24]

TABLE III
PERFORMANCE SUMMARY OF DESIGN VARIANTS ON ARTIX-7

Design	Freq, (MHz)	HAWK-512					HAWK-1024				
		Sign Cycles	Verify Cycles	Total Cycles	Latency (ms)	ATP	Sign Cycles	Verify Cycles	Total Cycles	Latency (ms)	ATP
(0)	**100**	4,374,898	2,144,755	6,519,653	65.197	481 (1×)	9,635,524	4,621,756	14,257,280	142.573	1051 (1×)
(I)	**100**	1,538,941	1,796,411	3,335,352	33.354	340 (0.708×)	3,349,600	3,891,649	7,241,249	72.412	739 (0.703×)
(II)	99	1,220,616	1,105,303	2,325,919	**23.494**	**289** (0.600×)	2,616,654	2,346,190	4,962,844	**50.130**	**616** (0.586×)
(III)	74	1,220,616	890,362	2,110,978	28.527	388 (0.808×)	2,616,654	1,866,516	4,483,170	60.583	824 (0.784×)
(IV)	68	**1,127,283**	**839,084**	**1,966,367**	28.917	408 (0.850×)	**2,413,942**	**1,751,178**	**4,165,120**	61.252	865 (0.823×)

Note: Area-Time-Product (ATP) = Equivalent Slice Utilization × Total Latency (in seconds) for HAWK-Sign and HAWK-Verify

TABLE IV
COMPARISON OF PROPOSED IMPLEMENTATION WITH STATE-OF-THE-ART RISC-V-BASED HARDWARE-SOFTWARE CO-DESIGN OF HAWK

Design	FPGA Platform	Design Version	FPGA Resource Utilization LUT / FF / DSP / BRAM	Algorithm Parameter Set	Freq. (MHz)	Sign Cycles	Verify Cycles	Sign Latency (ms)	Verify Latency (ms)
Karl *et al.*, 2024 [25]	Zynq UltraScale+	CV32E40P RISC-V + Keccak Accel + NTT Accel	22.4k / 13.2k / 13 / 54	Dilithium-V	–	4,357,249	1,848,324	–	–
				FALCON-1024		–	613,911	–	–
Ye *et al.*, 2025 [21]	Zynq-7000	CV32E40P RISC-V Baseline	10.8k / 3.3k / 5 / 24	HAWK-512	155	3,185,113	2,531,985	20.55	16.34
				HAWK-1024		6,625,515	5,159,475	42.75	33.29
		CV32E40P RISC-V + SIMD Co-Processor	30k / 10k / 37 / 24	HAWK-512	125	404,481	311,522	3.24	2.49
				HAWK-1024		840,335	624,621	6.72	5.00
This Work (II)	Artix-7	Vex RISC-V Baseline	1.9k / 1.8k / 4 / 27	HAWK-512	100	4,374,898	2,144,755	43.75	21.45
				HAWK-1024		9,635,524	4,621,756	96.35	46.22
		Vex RISC-V + Custom Function Unit	11.7k / 6.3k / 15 / 30	HAWK-512	99	1,220,616	1,105,303	12.33	11.16
				HAWK-1024		2,616,654	2,346,190	26.43	23.70

degraded critical path delay. To fairly analyze the design trade-offs between these variants, we compare their equivalent slice utilization [24], total signature computation and verification latency and area-time-product (equivalent slice utilization × total latency). Clearly, all our design variants achieve better latency and area-time-product than the baseline. In particular, Design II is the best in terms of latency and area-time-product, while Design IV achieves the best cycle count.

Table IV compares the FPGA resource utilization and performance of our best variant (Design II) with state-of-the-art hardware-software co-designs for lattice-based digital signature schemes. Being a new signature scheme, HAWK implementations in literature are very limited. The implementation from [21] uses the performance-oriented CV32E40P RISC-V core with SIMD optimizations and co-processors for HAWK signature generation and verification to achieve low latency at the cost of large resource overhead. Our CFU-enhanced Design II achieves significantly lower resource utilization compared to [21] at the cost of increased latency. The implementation from [25] demonstrates flexible hardware accelerators interfaced with a CV32E40P core to accelerate Dilithium and FALCON. A fair comparison is difficult due to different underlying FPGA platforms and limited performance metrics available in [25]. Our Design II achieves a lower total cycle count for signature generation and verification at NIST-V security level with HAWK-1024 compared to Dilithium-V [25]. While [25] achieves lower cycle count for performing FALCON-1024 signature verification, it does not support FALCON signature generation which requires floating-point arithmetic. Compared to our baseline, Design II achieves $\approx 3\times$ speedup with $\approx 67\%$ overhead, while maintaining overall low resource utilization suitable for constrained IoT applications.

V. CONCLUSIONS AND FUTURE WORK

HAWK is the only lattice-based scheme in Round 3 of the "Additional Digital Signatures" phase of the NIST PQC Standardization process. HAWK's compact key and signature sizes and simple design without floating point operations make it attractive for IoT applications. We demonstrate a light-weight software-hardware co-design of HAWK signature computation and verification using CFUs for Keccak permutation and polynomial transformations integrated with the open-source Vex RISC-V core. Our implementation supports both HAWK-512 and HAWK-1024 at different NIST security levels. We present multiple CFU design variants with varying degrees of acceleration to analyze the trade-offs between resource overheads and performance gains. Our best design achieves $\approx 3\times$ speedup and $\approx 40\%$ reduction in area-time-product compared to the baseline when implemented on a Xilinx Artix-7 FPGA. Our implementation is highly resource-efficient compared to previous SIMD-accelerated designs, thus making it appropriate for constrained embedded systems. Extension of our design to other RISC-V cores, side-channel analysis and ASIC implementations will be explored in future work.

ACKNOWLEDGMENT

This work was supported in part by the ABB Research Award in Honor of Hubertus von Grünberg and in part by the Centre of Excellence in Cyber Security (CySecK), Government of Karnataka. The authors would like to thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM Journal of Computing*, vol. 26, no. 5, pp. 1484–1509, 1997.

[2] L. Chen *et al.*, "Report on Post-Quantum Cryptography," NIST IR 8105, 2016.

[3] G. Alagic *et al.*, "Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process," NIST IR 8240, 2019.

[4] ——, "Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process," NIST IR 8309, 2020.

[5] ——, "Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process," NIST IR 8413, 2022.

[6] ——, "Status Report on the Fourth Round of the NIST Post-Quantum Cryptography Standardization Process," NIST IR 8545, 2025.

[7] S. Bai *et al.*, "CRYSTALS-Dilithium – Algorithm Specifications and Supporting Documentation," 2021.

[8] P. A. Fouque *et al.*, "Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU," 2020. [Online]. Available: https://falcon-sign.info/falcon.pdf

[9] G. Alagic *et al.*, "Status Report on the First Round of the Additional Digital Signature Schemes for the NIST Post-Quantum Cryptography Standardization Process," NIST IR 8528, 2024.

[10] ——, "Status Report on the Second Round of the Additional Digital Signature Schemes for the NIST Post-Quantum Cryptography Standardization Process," NIST IR 8610, 2026.

[11] L. Ducas *et al.*, "HAWK: Module LIP Makes Lattice Signatures Fast, Compact and Simple," in *Advances in Cryptology (ASIACRYPT)*, 2022, pp. 65–94.

[12] J. W. Bos *et al.*, "HAWK: A Signature Scheme inspired by the Lattice Isomorphism Problem," 2025. [Online]. Available: https://hawk-sign.info/hawk-spec.pdf

[13] "Vex RISC-V: A FPGA Friendly 32 bit RISC-V CPU Implementation." [Online]. Available: https://github.com/SpinalHDL/VexRiscv

[14] "The CFU Playground: Accelerate ML models on FPGAs." [Online]. Available: https://cfu-playground.readthedocs.io/en/latest/index.html

[15] S. Prakash *et al.*, "CFU Playground: Full-Stack Open-Source Framework for Tiny Machine Learning (TinyML) Acceleration on FPGAs," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 157–167.

[16] A. Waterman *et al.*, "The RISC-V Instruction Set Manual." [Online]. Available: https://github.com/riscv/riscv-isa-manual

[17] NIST, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," FIPS 202, 2015.

[18] D. J. Bernstein, "Fast Multiplication and its Applications," *Algorithmic Number Theory*, vol. 44, pp. 325–384, 2008.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[20] M. J. Kannwischer, M. Krausz, R. Petri, and S. Yang, "pqm4: Benchmarking NIST Additional Post-Quantum Signature Schemes on Microcontrollers," Cryptology ePrint Archive, Paper 2024/112, 2024. [Online]. Available: https://eprint.iacr.org/2024/112

[21] Z. Ye *et al.*, "PQNTRU: Acceleration of NTRU-Based Schemes via Customized Post-Quantum Processor," *IEEE Transactions on Computers*, vol. 74, no. 5, pp. 1649–1662, 2025.

[22] G. Bertoni *et al.*, "KECCAK Implementation Overview," 2012. [Online]. Available: https://keccak.team/files/Keccak-implementation-3.2.pdf

[23] R. Shrivastava *et al.*, "A Unified Hardware Accelerator for Fast Fourier Transform and Number Theoretic Transform," in *IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, 2025, pp. 1–5.

[24] D. Kundi *et al.*, "Ultra High-Speed Polynomial Multiplications for Lattice-Based Cryptography on FPGAs," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 4, pp. 1993–2005, 2022.

[25] P. Karl *et al.*, "Post-Quantum Signatures on RISC-V with Hardware Acceleration," *ACM Trans. Embed. Comput. Syst.*, vol. 23, no. 2, 2024.