VAXEN: A Versatile AVX Extension for Zero-Knowledge Proof and Post-Quantum Cryptography

Jipeng Zhang*, Hao Cheng†‡✉, Tao Lu*, Jiaheng Zhang*✉
*School of Computing, National University of Singapore, Singapore
†School of Cyber Science and Technology, Shandong University, Qingdao, China
‡State Key Laboratory of Cryptography and Digital Economy Security, Shandong University, Qingdao, China
jp-zhang@outlook.com, hao.cheng@sdu.edu.cn, lutaocc2020@gmail.com, jhzhang@nus.edu.sg

***Abstract*—**
The Intel Advanced Vector eXtension (AVX) family provides multiply-high instructions (e.g., vpmulhw) for 16-bit lanes, but no corresponding support exists for 32-bit lanes in either AVX2 or AVX-512. This gap inflates the cost of 32-bit integer arithmetic in both zero-knowledge proof (ZKP) and post-quantum cryptography (PQC), and it also impacts the efficiency of Philox pseudorandom number generation (PRNG). To address this, we present VAXEN, a Versatile AVX ExtensioN, which introduces signed and unsigned 32-bit multiply-high instructions, vpmulhd and vpmulhud, while preserving the existing AVX2/AVX-512 programming model. VAXEN targets per-lane arithmetic and therefore does not require carry-propagation support.

We evaluate VAXEN using proxy ISA (PISA): proxy binaries are measured on Intel Rocket Lake, Intel Core Ultra 5 228V P/E cores, and AMD Zen 5, then interpreted under explicit implementation mappings. Under a balanced 32-bit multiply mapping, projected results show up to $1.86\times$ (resp. $2.14\times$) speedup for BabyBear (resp. ML-DSA) Montgomery multiplication, $1.84\times$ for ML-DSA NTT, $1.51\times$ for Rust-level Plonky3 Poseidon2, and $2.83\times$ for Philox PRNG. We also report a complete Intel PISA matrix that separates the benefit of the proposed high-half instructions from improvements to existing vpmulld. To strengthen trustworthiness, we use CryptoLine to formally verify the projected implementations of Montgomery multiplication and Philox PRNG. We also develop an ASIC evaluation library and show low marginal cost: $+4.50\%$ of the modeled vector-integer cluster post-synthesis, 0.017% of an AMD Zen 5 core area including private L2, and a post-route core-normalized upper bound below 0.1%. These results suggest that a minimal fix to AVX2 and AVX-512 can deliver meaningful cross-domain speedups at low hardware cost.

***Index Terms*—ISA extension, AVX2, AVX-512, proxy ISA, post-quantum cryptography, zero-knowledge proofs**

I. Introduction

Recent cryptographic and proof workloads increasingly rely on arithmetic over 32-bit prime fields[1]. This trend appears in zero-knowledge proof (ZKP), where implementations built on PLONK [1] and Plonky3 [2] depend on fields such as BabyBear and KoalaBear. It also appears in post-quantum cryptography (PQC), where the NIST-standardized Module-Lattice-Based Digital Signature Algorithm (ML-DSA) [3] relies on 32-bit modular multiplication inside its Number Theoretic Transform (NTT) kernels. At the same time, software stacks use 32-bit high-half multiplication in Philox pseudo-random number generation (PRNG) [4], which is deployed in PyTorch and TensorFlow [5], [6]. These workloads are algorithmically diverse, but they share a common microarchitectural requirement: efficient Single Instruction, Multiple Data (SIMD) support for 32-bit multiply-high.

x86 SIMD currently exposes an asymmetric multiplication interface. For 16-bit lanes, AVX2 and AVX-512 provide both low-half (vpmullw) and high-half (vpmulhw) products. For 32-bit lanes, however, they only provide the low-half product (vpmulld) directly, while programmers must synthesize the high half using widening multiplies (vpmuludq), lane shuffles (e.g., vmovshdup), and blends (vpblendd). This asymmetry is not merely an instruction-count nuisance. It lengthens dependence chains, increases shuffle pressure, complicates library implementations, and makes an important class of 32-bit arithmetic kernels systematically inefficient on x86.

The impact of this gap is broad. In cryptography, it increases the cost of Montgomery multiplication [7] that dominates the inner loops of ZKP and PQC software—for example, as Figure 1 illustrates, ML-DSA Montgomery multiplication requires 12 vector instructions on the baseline path versus only 4 with our proposed VAXEN. In non-cryptographic software, the same missing primitive reappears in Philox PRNG. These domains are usually studied separately, yet they point to the same architectural conclusion: x86 lacks a versatile 32-bit SIMD multiply-high primitive that repeatedly appears in modern integer kernels.

This paper presents VAXEN, a versatile AVX extension for 32-bit multiply-high workloads. VAXEN adds signed and unsigned 32-bit multiply-high SIMD instructions, vpmulhd and vpmulhud, for AVX2 and AVX-512 while preserving the existing programming model and keeping the semantics strictly per lane. The design goal is deliberately modest: rather than proposing a large cryptography-specific extension, we

✉ Corresponding authors.

[1]We use "32-bit prime field" for moduli q with $2^{16} < q < 2^{32}$, where each SIMD lane naturally holds one field element.

target the narrow primitive that repeatedly appears across ZKP, PQC, and Philox PRNG. Because the target arithmetic is per lane rather than multi-word, the extension does not require carry propagation support.

To evaluate VAXEN, we use performance projection using proxy ISA (PISA) [8] on Intel Rocket Lake, the Lion Cove P-core and Skymont E-core of an Intel Core Ultra 5 228V (Lunar Lake, 2024) processor, and AMD Zen 5. We measure proxy binaries on real hardware and then interpret those measurements under explicit implementation mappings, rather than presenting the projections as direct hardware measurements of a nonexistent instruction. The cryptographic class spans three scales: arithmetic primitives such as Montgomery multiplication, cryptographic components such as ML-DSA NTT and Plonky3 Poseidon2 [9], and full ML-DSA signature generation. The non-cryptographic class is Philox PRNG, which is widely used in AI frameworks such as PyTorch and TensorFlow. Across these workloads, the same missing primitive consistently appears as a bottleneck.

Beyond performance projection, we also study trustworthiness and hardware cost. We use CRYPTOLINE [11] to formally verify the projected implementations of BabyBear and ML-DSA Montgomery multiplication and Philox PRNG, ensuring that the optimized arithmetic paths preserve the intended semantics and range contracts. We further develop an ASIC evaluation library for the proposed datapath and use it to estimate the hardware overhead of integrating the new operation. Our current prototype indicates that the added datapath cost is low, supporting the claim that VAXEN is a small architectural change with cross-domain benefits.

Our implementations are publicly available at https://github.com/Ji-Peng/VAXEN_Artifact under the MIT license.

We frame the paper around a broader architectural thesis: when a single missing lane-wise primitive recurs across multiple performance-critical domains, a minimal ISA patch can deliver disproportionate benefit relative to its hardware cost. We demonstrate this thesis through a multi-layered evaluation methodology—PISA-based performance projection, static microarchitectural analysis, formal verification, and ASIC cost estimation—that can serve as a template for evaluating other narrow ISA extensions. This paper makes the following contributions:

- **Cross-domain bottleneck identification.** We show that the absence of 32-bit SIMD multiply-high is a shared architectural bottleneck across ZKP, PQC, and Philox PRNG on x86, and we analyze the microarchitectural origins of this gap on Intel Rocket Lake, Intel Core Ultra 5 228V (Lion Cove/Skymont), and AMD Zen 5 to argue that existing multiplier organizations can support the proposed instructions with minimal modification (Section II).
- **Minimal ISA design with design-space justification.** We propose VAXEN, a minimal AVX2 and AVX-512 extension that adds signed and unsigned 32-bit multiply-high instructions with per-lane semantics, and we explicitly justify this design point against more aggressive alternatives such as fused reduction instructions or carry-propagating extensions (Section III).
- **Performance projection with sensitivity.** We evaluate VAXEN using PISA on Intel Rocket Lake, Intel Core Ultra 5 228V (Lion Cove/Skymont), and AMD Zen 5. The main balanced projection reaches up to $2.14\times$ for Montgomery multiplication, $1.84\times$ for ML-DSA NTT, and $2.83\times$ for Philox PRNG, while a complete Intel PISA matrix separates the proposed high-half instructions from improvements to existing vpmulld (Section V).
- **Trustworthiness and hardware cost.** We formally verify the projected Montgomery-multiplication and Philox implementations with CRYPTOLINE and synthesize the datapath, which costs $+4.50\%$ of the modeled vector-integer cluster post-synthesis and below 0.1% of a Zen 5 core area including private L2 under a post-route upper bound (Section VII).

Taken together, these results argue that fixing x86's 32-bit SIMD multiply-high gap is not a niche cryptographic tweak, but a versatile improvement that benefits a wider class of modern integer kernels. More broadly, the paper illustrates how a combination of proxy-ISA projection, formal verification, and ASIC prototyping can build a credible case for minimal ISA changes without access to proprietary microarchitectural simulators.

II. BACKGROUND AND MOTIVATION

This section establishes the architectural problem that motivates VAXEN. We first summarize the relevant Montgomery arithmetic background, then analyze the 32-bit SIMD Montgomery multiplication to expose the algorithmic origin of the 32-bit gap, and finally connect that contrast to existing x86 SIMD instruction support and the scope of the proposed extension.

A. Notations

We use $\bmod$ to denote congruence modulo an integer. For $z \in \mathbb{Z}$, we write $z \bmod^{\pm} q \in \{-\lfloor \frac{q}{2} \rfloor, -\lfloor \frac{q}{2} \rfloor + 1, \ldots, \lfloor \frac{q-1}{2} \rfloor\}$ for the canonical signed representative of z modulo q, and $z \bmod^{+} q \in \{0, 1, \ldots, q-1\}$ for the canonical unsigned representative.

B. Montgomery Multiplication

Two Montgomery-style formulations are especially relevant to this paper: original Montgomery multiplication [7] and Seiler's variant [12]. Let β denote the radix such that q fits in one word, for example $\beta = 2^{32}$ when $2^{16} < q < 2^{32}$.

Let a and b denote the two multiplicative inputs. The original Montgomery multiplication computes

$$r = \frac{ab + \left((ab(-q^{-1} \bmod \beta)) \bmod^{\pm} \beta\right) q}{\beta}.$$

Seiler's variant computes

$$\begin{aligned} r &:= \mathbf{MontMul}_q(a, b) \\ &= \frac{ab - \left((ab(-q^{-1} \bmod \beta)) \bmod^{\pm} \beta\right) q}{\beta}. \end{aligned}$$

The practical advantage of Seiler's variant is that the numerator is divisible by β, so the low-half subtraction can

ML-DSA 32-bit MontMul (Baseline) [10]

```
ao       = vmovshdup(a)
me,mo,ce = vpmuldq(a,bqe),vpmuldq(ao,bqo), vpmuldq(a,be)
co,te,to = vpmuldq(ao,bo),vpmuldq(me,q),   vpmuldq(mo,q)
ch = vpblendd(0xAA, co, vmovshdup(ce))
th = vpblendd(0xAA, to, vmovshdup(te)), r = vpsubd(ch,th)
```

ML-DSA MontMul with VAXEN

```
ml = vpmulld(a, bqi)
ch = vpmulhd(a, b)
th = vpmulhd(ml, q)
r  = vpsubd(ch, th)
```

Fig. 1. SIMD Montgomery multiplication for ML-DSA over $q = 8380417 < 2^{32}$. Left: the baseline AVX2 path requires 12 vector instructions due to the missing 32-bit multiply-high; the known twiddle factor b allows precomputing constants, where be (bo) denotes the even (odd) channel values of b, and bqe (bqo) denotes the corresponding even (odd) channel values of bqi = $b \cdot (-q^{-1} \bmod \beta) \bmod^{\pm} \beta$. Right: with VAXEN, the same reduction completes in 4 instructions.

be ignored safely when only the upper half is carried forward. In contrast, the original formulation must still account for carry propagation from the low half into the high half. Consequently, Seiler's approach has been widely adopted in standard implementations, including ML-KEM [13] and ML-DSA [3]. Throughout this work, all instances of Montgomery multiplication follow this methodology.

If we denote m by

$$m := \left((ab(-q^{-1} \bmod \beta)) \bmod^{\pm} \beta\right),$$

then $|m| \le q/2$, and the output magnitude of both formulations satisfies

$$|r| \le \frac{|ab|}{\beta} + \frac{q}{2}.$$

C. Algorithmic Origin of the 32-bit Gap

Figure 1 contrasts ML-DSA Montgomery multiplication on the baseline AVX2 path (12 vector instructions [10]) with the proposed VAXEN path (4 instructions). The baseline must decompose the reduction into even- and odd-lane widening-multiply-plus-repack shapes, tripling the instruction count and lengthening dependence chains through vmovshdup and vpblendd repairs.

D. Existing x86 SIMD Instructions

AVX2 and AVX-512 provide low-half and high-half products for 16-bit lanes but lack high-half products for 32-bit lanes. The workaround uses even-lane widening multiplies (vpmuldq and vpmuludq), lane-repair shuffles (vmovshdup), and blends (vpblendd), as shown in Figure 1. Throughout this paper, AVX2 and AVX-512 forms share the same mnemonics; we refer to the AVX2 form for ymm operands and AVX-512 for zmm.

Table I summarizes AVX2 performance on Rocket Lake, Lion Cove, Skymont, and Zen 5. Two trends matter: Rocket Lake exposes the historical Intel imbalance (vpmulld at 10/1 versus 16-bit multiplies at 5/0.5), while Lion Cove/Skymont reduce the vpmulld latency to 6/1 and 4/1 but still leave its reciprocal throughput behind the 16-bit family. On AMD, the 16-bit and existing 32-bit multiply families are balanced from Zen 3 through Zen 5 [14].

Microarchitectural Analysis. We now present a microarchitectural analysis informed by the observed performance data in Table I. While we do not have access to proprietary design details, the following reasoning is consistent with published instruction-level measurements [14] and provides the architectural basis for VAXEN's feasibility argument. We do not consider reuse of integer multiplication inside floating-point execution engines.

TABLE I
AVX2 INSTRUCTION LATENCY/THROUGHPUT.

Instruction	Rocket	Lion Cove	Skymont	Zen 5
vpmullw	5/0.5	5/0.5	3/0.5	3/0.5
vpmulhw	5/0.5	5/0.5	3/0.5	3/0.5
vpmulhuw	5/0.5	5/0.5	3/0.5	3/0.5
vpmulld	10/1	6/1	4/1	3/0.5
vpmuldq	5/0.5	4/0.5	3/0.5	3/0.5
vpmuludq	5/0.5	4/0.5	3/0.5	3/0.5
vmovshdup	1/0.5	1/0.5	1/0.5	1/0.25
vpblendd	1/0.33	1/0.25	1/0.5	1/0.25

Note: Values are latency/CPI. Rocket and Zen 5 are from [14]; Lion Cove and Skymont are measured on an Intel Core Ultra 5 228V (Lunar Lake) with Turbo Boost and Hyper-Threading disabled.

For Rocket Lake, the vpmuldq/vpmuludq latency/CPI of $5/0.5$ is consistent with two independent 256-bit pipelines, each containing four parallel 32x32→64 multipliers organized as a 5-stage pipeline. The vpmulld point of $10/1$ is the anomalous entry: its latency is exactly $2\times$ that of vpmuludq, and its CPI is exactly $2\times$ worse, which strongly suggests that vpmulld is not a first-class hardware operation but is instead decomposed internally into two sequential vpmuludq-class micro-operations that process even and odd 32-bit lanes separately and then merge the low halves. If this hypothesis is correct, it carries an important implication for VAXEN: implementing vpmulhd/vpmulhud on Intel hardware would not require adding new multiplier datapaths. Since the existing even-lane 32x32→64 multipliers already produce a full 64-bit product, reading the upper 32 bits requires only a result selection multiplexer (MUX). In other words, the same two-micro-operation decomposition that Intel currently uses for vpmulld could serve vpmulhd at the same latency 10 and CPI 1, with near-zero additional silicon cost.

For Zen 5, the uniform $3/0.5$ behavior across the 32-bit multiply family is consistent with a design built from dual-mode macrocells: in full mode, one macrocell computes one 32x32→64 product in 3 stages, while in low mode it computes two 32x32→low-half products in the same 3-stage pipeline.

One plausible organization is therefore two independent 256-bit pipelines, each containing four parallel dual-mode macrocells. On this design line, adding vpmulhd/vpmulhud at full throughput (latency 3, CPI 0.5) requires upgrading the low-only lanes to full 64-bit products, which is exactly the change evaluated in our ASIC study (Section VII).

In summary, the observed performance data suggests that both Intel and AMD could support the proposed VAXEN instructions within their existing multiplier organizations, but with different performance profiles. An AMD-like balanced design would deliver CPI 0.5 with a small area increment. An Intel-like decomposition approach would deliver CPI 1.0 with near-zero area cost, and the measurements of the Lion Cove and Skymont cores show that newer Intel cores are already reducing the latency of the 32-bit multiply penalty. Our main PISA projection assumes the balanced path; the Intel PISA matrix in Section IV reports measured sensitivity points for less balanced choices.

E. Where the Gap Appears

We only summarize these workloads at a high level here to show where the missing primitive arises; further algorithmic and implementation details are deferred to the corresponding references.

Cryptographic scenarios. In ML-DSA [3], the NTT uses Cooley–Tukey butterflies [15] of the form $(a, b) \mapsto (a+b, a-b \cdot w)$, where the product $b \cdot w$ is implemented with the 32-bit Montgomery multiplication shown in Figure 1; this makes the butterfly a direct consumer of the missing lane-wise 32-bit multiply-high primitive. The same arithmetic bottleneck also propagates to full ML-DSA signature generation, because signing repeatedly invokes the NTT-centered kernels inside a user-visible end-to-end workload (e.g., post-quantum TLS 1.3 handshake [16], [17]). In ZKP systems, Poseidon2 [9] is a representative permutation-based workload: its parameter w denotes the permutation state width, i.e., the number of field elements in the state, so the later $w = 16$ and $w = 24$ cases correspond to 16- and 24-element states. In Plonky3 [2] over the BabyBear field $q = 2^{31} - 2^{27} + 1$, one key nonlinear component is the S-box exponentiation (e.g., $x \mapsto x^7$), whose modular multiplications repeatedly exercise the same 32-bit Montgomery path.

Non-cryptographic scenario. Philox [4] is a counter-based PRNG used in software stacks such as PyTorch and TensorFlow [5], [6]. A Philox round repeatedly forms products of the form $M \cdot x = 2^{32} \cdot \mathrm{hi}(M, x) + \mathrm{lo}(M, x)$ and mixes the high half into the next counter state, so computing $\mathrm{hi}(M, x) = \lfloor Mx/2^{32} \rfloor$ is part of the hot path.

Across these cryptographic and non-cryptographic kernels, the missing operation is therefore both narrow and versatile: it is a single lane-wise arithmetic primitive, but it reappears in several performance-critical workloads.

III. VAXEN ISA Design

This section defines the proposed extension. VAXEN is not a broad cryptographic extension suite; it is a lane-wise extension that supports the 32-bit multiply-high primitive in AVX2 and AVX-512.

TABLE II
Proposed VAXEN instruction encoding.

Instruction	Map	VEX opcode	EVEX opcode
vpmulhud	66.0F38	WIG48 /r	W048 /r
vpmulhd	66.0F38	WIG49 /r	W049 /r

Note: Both instructions use the 66.0F38 opcode map with NDS form; opcode bytes 48 and 49 are selected from entries marked unassigned in the SDM/XED revisions [18].

A. Instruction Set Scope

VAXEN adds two core operations: signed 32-bit multiply-high, vpmulhd, and unsigned 32-bit multiply-high, vpmulhud. Each operation is provided in AVX2 and AVX-512 forms so that the proposal applies uniformly to the two dominant x86 SIMD programming models used by today's optimized software. The semantic contract is simple: for each 32-bit lane, the instruction returns the upper 32 bits of the 64-bit product, yielding 8-way parallel 32x32 multiply-high in AVX2 and 16-way parallel 32x32 multiply-high in AVX-512.

The proposed instructions are defined independently for each lane. This is an important distinction from multi-word cryptographic extensions (e.g., MQX [8]), where carry propagation or cross-word interactions are part of the ISA contract. The kernels targeted in this paper do not need those semantics. For BabyBear arithmetic, ML-DSA arithmetic, and Philox PRNG, the key operation is the high half of a lane-wise product.

B. Instruction Encoding

VAXEN requires only two new opcodes and fits within the existing x86 VEX/EVEX (Vector Extension / Enhanced Vector Extension) prefix framework without requiring a new opcode map. As shown in Table II, both instructions are placed in the 66.0F38 map alongside the existing 32-bit multiply family, with opcode bytes 48 and 49 proposed from entries marked unassigned in the SDM/XED revisions [18]. The VEX forms follow the WIG convention of AVX2 32-bit integer multiplies; the EVEX forms set W=0 for this instruction class and inherit standard AVX-512 opmask and zeroing-masking semantics. Memory broadcast can be enabled if the final EVEX tuple definition includes it. A dedicated CPUID feature bit gates both instructions for the two targeted vector widths (256-bit VEX and 512-bit EVEX), following the same enumeration model as AVX-VNNI and AVX-IFMA.

C. Constant-Time Guarantee

Cryptographic software requires data-independent execution time to prevent timing side-channel attacks. The ISA contract requires vpmulhd and vpmulhud to execute in constant time: the latency and micro-operation count are fixed regardless of operand values. The evaluated datapath satisfies this contract (Section VII): the Booth radix-4 encoder, Wallace-tree

TABLE III
MQX AND VAXEN ON MONTGOMERY MULTIPLICATION.

Design	Granularity	ML-DSA MontMul
AVX2 baseline	32-bit lanes	12 vector inst.
MQX-style [8]	64-bit lanes	≥ 6 vector inst.
VAXEN	32-bit lanes	4 vector inst.

Note: Counts are for eight coefficients; the MQX-style row is an optimistic lower bound without unpacking and repacking.

TABLE IV
PROXY INSTRUCTIONS USED BY THE PISA MATRIX.

Target instruction	Proxy instruction
vpmulld	vpmullw
vpmulhd	vpmulhw
vpmulhud	vpmulhuw

Note: vpmulld is an existing x86 instruction; Table VI isolates the effect of keeping it at the measured cost of each Intel core.

TABLE V
EVIDENCE BOUNDARY USED IN THE EVALUATION.

Evidence	Status	Boundary
Base cycles	Measured	Existing AVX cost
Proxy binaries	Measured proxy	Substituted instruction class
Speedup	Derived	Mapping-dependent projection
llvm-mca	Static	Supported-core scheduling only
CryptoLine	Proof	Functional correctness only
RTL/PnR	Hardware model	Area/timing plausibility

Note: Proxy-binary rows are measured on real hardware, but projected VAXEN speedups are derived by interpreting those proxy cycles under the PISA mapping in Table IV; they are not direct hardware measurements of nonexistent instructions.

compressor, and prefix adder are all combinational structures whose delay depends on operand width, not values. The MulHiSignedCorrector applies an unconditional correction for every lane on every invocation, selected by opcode rather than data. This is the same principle that makes existing vpmulhw, vpmulhuw, vpmuludq, and vpmuldq constant-time on current Intel and AMD processors. VAXEN introduces no data-dependent branching, variable-latency units, or operand-dependent power gating, so it does not open new timing side channels beyond those in the baseline multiply datapath. As with existing SIMD multipliers, power and electromagnetic side channels due to operand-dependent Hamming weight in the Booth encoder remain orthogonal and require platform-level countermeasures.

D. Design Space Exploration

VAXEN deliberately occupies the minimal end of the ISA-extension design space. We considered and rejected three more aggressive alternatives.

Fused Montgomery reduction instruction. Fusing the low-half multiply, high-half multiply, and subtraction into a single three-source instruction could reduce Figure 1 (right) from 4 to 2 instructions. However, three-source SIMD instructions require wider register-file read ports and more complex scheduling, and a fused reduction would be Montgomery-specific, providing no benefit to Philox PRNG. VAXEN's multiply-high primitive is the common factor across all target workloads.

Carry-propagating multi-word extension. Extensions such as MQX [8] introduce carry-aware multi-word operations for large-integer kernels. Our target workloads operate entirely within single 32-bit lanes and do not require cross-lane carry propagation. ML-DSA Montgomery multiplication makes the difference concrete. Packing two 32-bit coefficients into one MQX 64-bit word is not correct because the product contains cross terms between the two coefficients. A correct MQX-style adaptation must widen each coefficient to a separate 64-bit word, halving lane density and adding shifts and repacking. Table III summarizes this instruction-level lower bound: MQX is complementary to VAXEN, but it does not remove the lane-wise 32-bit high-half gap. We do not compare area because MQX does not report a directly comparable RTL/PnR cost for this ML-DSA design point.

Wider element support (64-bit multiply-high). A $64 \times 64 \rightarrow 128$ multiply-high would benefit large-integer arithmetic but is unnecessary for 32-bit prime fields in ZKP and PQC, and the multiplier area cost would be substantially larger.

In summary, by targeting the narrowest missing primitive that recurs across multiple domains, VAXEN maximizes its versatility-to-cost ratio while remaining compatible with existing SIMD programming models.

IV. EVALUATION METHODOLOGY

This section describes how the paper evaluates VAXEN. The methodology is multi-layered: PISA-based performance projection [8], static scheduling analysis, formal verification, and ASIC-oriented cost estimation.

A. Workload Categories

The evaluation is organized around two workload categories. The first is cryptographic, which we study at three scales: Montgomery multiplication [7], [12], cryptographic components such as ML-DSA NTT [3], [10] and Plonky3 Poseidon2 [2], [9], and full ML-DSA signature generation. The second is non-cryptographic and centers on Philox PRNG [4], a multiply-high-intensive kernel widely used in AI frameworks such as PyTorch and TensorFlow. Table VII includes workloads for which we have end-to-end or component-level benchmarks. Table VIII instead uses extracted hot snippets to explain instruction count, block throughput, and critical path; it is diagnostic evidence, not a replacement for the measured proxy benchmarks.

B. PISA-Based Performance Projection

The primary performance methodology is proxy ISA (PISA) [8]. The basic idea is to estimate a proposed instruc-

tion's performance by mapping it to an existing instruction that is structurally similar on real hardware, rather than trying to reconstruct proprietary Intel or AMD microarchitectures in detail. If the new instruction were implemented efficiently, its latency, throughput, and port usage would then be expected to track those of its proxy instruction closely enough to support early-stage architectural evaluation.

As shown in Table IV, we use 16-bit multiplication instructions to proxy 32-bit multiplication instructions. Our main projection corresponds to a balanced implementation hypothesis in which future 32-bit multiply-high instructions (i.e., vpmulhd and vpmulhud), alongside the existing 32-bit multiply-low instruction (i.e., vpmulld), approach the latency and CPI of mature 16-bit SIMD multiply instructions. This is a mapping assumption for performance projection, not a direct measurement of nonexistent VAXEN hardware; Table V states this boundary explicitly.

To address the central concern that the balanced mapping might conflate the proposed high-half instruction with a faster vpmulld, we also run a complete 2x2 PISA matrix on real Intel hardware. Holding the compact VAXEN dataflow fixed, we independently vary the low-half multiply and the high-half multiply between the measured vpmulld cost of the target core and the 16-bit-multiply proxy cost. This yields four proxy points: all-vpmulld, fast high only, fast low only, and balanced.

Table VI reports this proxy-isolation experiment for ML-DSA NTT/INTT on Rocket Lake, Lion Cove, and Skymont cores. The high-only rows isolate the performance attributable to the high-half operation under the stated PISA mapping while vpmulld remains unchanged; they improve over baseline by $1.29\times$–$1.79\times$ across the three Intel cores. The all-mulld rows are also informative: they are modest on Rocket Lake and Lion Cove, but much stronger on Skymont because the vpmulld latency is substantially lower than Rocket Lake's. Thus the measured matrix shows both halves of the story: VAXEN's proposed high-half operation is an independent source of benefit, and Intel's newer cores are already reducing the historical penalty of 32-bit vector multiplication.

TABLE VI
ML-DSA CONTRIBUTION ISOLATION ON INTEL CORES.

Kernel	Proxy	Rocket	Lion Cove	Skymont
NTT	baseline	918	1130	2189
	all-mulld	864/1.06	1091/1.04	1267/1.73
	high-only	690/1.33	844/1.34	1221/1.79
	low-only	738/1.24	942/1.20	1200/1.82
	balanced	580/1.58	666/1.70	1187/1.84
INTT	baseline	808	940	2193
	all-mulld	726/1.11	930/1.01	1330/1.65
	high-only	582/1.39	727/1.29	1243/1.76
	low-only	622/1.30	783/1.20	1227/1.79
	balanced	482/1.68	571/1.65	1177/1.86

Note: Non-baseline cells report cycles/speedup over baseline. all-mulld: vpmulld proxies every 32-bit multiply; high-only: vpmulhw/vpmulhuw proxy vpmulhd/vpmulhud; low-only: vpmullw proxies vpmulld, and vpmulld proxies vpmulhd/vpmulhud; balanced: both 16-bit substitutions are used.

C. Validity of the PISA Assumption

The balanced proxy mapping assumes that the proposed vpmulhd and vpmulhud instructions can approach the latency and CPI of today's 16-bit multiply-high instructions. This subsection argues that the assumption is grounded in empirical trends and our own hardware evaluation, while the Intel proxy matrix in Table VI explicitly bounds less balanced choices.

Empirical evidence from AMD. On AMD Zen 3 through Zen 5, all existing 32-bit integer multiply instructions (vpmulld, vpmuldq, vpmuludq) already achieve latency 3 and CPI 0.5, identical to the 16-bit multiply family [14]. This demonstrates that latency/CPI parity between existing 16-bit and 32-bit vector multiply operations is already shipped; the missing high-half operation itself remains the proposed ISA addition.

Hardware timing validation. Our ASIC evaluation library (Section VII) provides direct evidence that the proposed data-path does not degrade timing. Both the baseline configuration (4 full + 4 low-only lanes) and the VAXEN configuration (8 full lanes with signed correctors) meet the same 1 GHz frequency target, with identical worst-case register-to-register arrival of 908 ps. The critical path in both cases is the Wallace-tree plus 64-bit prefix-adder chain inside FullMul32Lane, which is already present in the baseline for even-lane vpmuludq operations. Adding vpmulhd/vpmulhud therefore does not introduce a new critical path; it replicates an existing one to cover the odd lanes.

Scope of the assumption. The PISA proxy does not claim that every possible x86 implementation will achieve CPI 0.5 for the new instructions. As discussed in Section VII, a minimal MUX-only implementation that reuses only even-lane multipliers would deliver CPI ≈ 1.0 instead. Our main projection targets a balanced-throughput design point consistent with AMD's existing 32-bit multiplies. On Intel, Table I shows a favorable trend: Rocket Lake has vpmulld latency/CPI of 10/1, while our Ultra 5 228V measurements show 6/1 on Lion Cove and 4/1 on Skymont. The reciprocal throughput remains less balanced than the 16-bit family, so the Intel proxy matrix should be read as a measured sensitivity study rather than as a claim that all Intel cores already match the balanced design point.

Control-flow and data-flow preservation. The PISA proxy replaces 32-bit multiply instructions with 16-bit counterparts, which produces numerically incorrect intermediate values but must preserve the program's control flow and instruction-level data flow for the timing projection to be valid. We verified this property by disassembling all proxy binaries and confirming that the compiler emits identical instruction sequences (same instruction count, operand widths, register allocation, and branch structure) except for the substituted multiply mnemonics. Because the substituted 16-bit multiplies

share the same operand format and encoding width as their 32-bit targets, no spill, branch, or scheduling divergence is introduced.

D. Static Analysis, Formal Verification, and ASIC Cost

For static analysis, we utilize llvm-mca [19] to evaluate execution port pressure and critical path latency. To guarantee the functional correctness of our PISA-based implementations, we formally verify the projected implementations of Montgomery multiplication and Philox PRNG using the CRYPTOLINE framework [11]. Furthermore, to assess the hardware area overhead of the proposed VAXEN extensions, we developed and synthesized an ASIC prototype.

V. PROJECTED PERFORMANCE EVALUATION

This section reports projected performance using the PISA methodology [8] on four measured cores/platforms: Intel Core i7-11700K (Rocket Lake), the Lion Cove P-core and Skymont E-core of an Intel Core Ultra 5 228V (Lunar Lake) processor, and AMD EPYC 9B45 (Zen 5, Google Cloud). We organize results into cryptographic kernels and a non-cryptographic Philox kernel to show how the proposed 32-bit multiply-high primitive performs under the balanced mapping, with the implementation sensitivity bounded in Section IV.

A. Experimental Setup

On the Rocket Lake and Core Ultra 5 228V platforms, Turbo Boost and Hyper-Threading were disabled to ensure deterministic measurements. The Core Ultra 5 228V experiments pin execution to CPU 0 for Lion Cove and CPU 4 for Skymont. For the Google Cloud-hosted Zen 5 platform, we ran the full suite on an instance configured with one thread per physical core.

We report median cycle counts. "Base" denotes existing AVX2 and AVX-512 implementations. "VAXEN" denotes the balanced projected implementation with proxy instructions according to Table IV; the complete Intel PISA matrix is reported separately in Table VI. All speedups are intra-ISA (AVX2 baseline vs. AVX2 VAXEN, AVX-512 baseline vs. AVX-512 VAXEN). The Core Ultra 5 228V cores support AVX2 but not AVX-512.

B. Montgomery Multiplication Kernels

Our implementation of the BabyBear and ML-DSA Montgomery multiplication is based on the vectorized routines in the Plonky3 [20], [21] and the Dilithium repository [10], respectively.

Table VII (upper rows) shows proxy-binary arithmetic evidence under the balanced PISA mapping. Notable primitive-level gains appear in BabyBear and ML-DSA Montgomery multiplication, especially on Skymont, where the balanced mapping improves ML-DSA Montgomery multiplication by $2.14\times$. AVX-512 remains available on Rocket Lake and Zen 5, while the Core Ultra 5 228V cores are AVX2-only.

Applicability to KoalaBear. KoalaBear ($q = 2^{31} - 2^{24} + 1$) is another 31-bit prime field used alongside BabyBear in the Plonky3 stack [2]. In Plonky3, the KoalaBear Montgomery multiplication follows the same vectorized instruction sequence as the BabyBear variant: both fields use identical SIMD packing, and identical widening-multiply-plus-repack shape on the baseline path. Therefore, because the SIMD packing and widening/repack shape match BabyBear, we expect similar speedups for KoalaBear; we do not separately benchmark it here.

C. Cryptographic Building Blocks

The ML-DSA NTT routines are adapted from the hand-optimized AVX2 assembly implementation provided in the Dilithium repository [10]. We report only AVX2 results for ML-DSA NTT because the official Dilithium repository maintains only an AVX2 implementation and does not provide an AVX-512 variant. For Poseidon2, we derived our PISA versions by directly extending the original Rust-based AVX2 and AVX-512 implementations [22], [23]. NTTs are included because each butterfly unit directly invokes Montgomery multiplication; Poseidon2 is included because its S-box layer relies on repeated BabyBear field multiplications, which contribute substantially to the arithmetic cost.

As shown in Table VII, ML-DSA NTT/INTT (AVX2) retain most of the primitive-level advantage ($1.56\times$–$1.86\times$ across the Intel cores) because the butterfly is dominated by Montgomery multiplication. Rust-level Plonky3 Poseidon2 is more bounded but still positive ($1.30\times$–$1.51\times$): its S-box benefits from faster Montgomery multiplication, but the Maximum Distance Separable layer and surrounding Rust-level code are unaffected by VAXEN.

BabyBear NTT and Plonky3 NTT. We also evaluate BabyBear NTT, a Cooley–Tukey NTT vectorized across butterfly indices, and Plonky3 NTT, the Rust-based batch DFT [24] that vectorizes across columns (width 32). BabyBear NTT achieves $1.41\times$–$1.55\times$ speedup, while Plonky3 NTT shows smaller, platform-sensitive gains because the benchmark is memory-system sensitive; VTune attributes 80.4% of baseline pipeline slots to memory stalls.

Prevalence in end-to-end proofs. To quantify prevalence inside a complete proof, we profiled a Plonky3 BabyBear Poseidon2 STARK proof (the prove_prime_field_31 example) on Rocket Lake. In the $\log_2$ trace-length-18 perf-symbol profile, Poseidon2 accounts for 65.2% of sampled CPU time and DFT/NTT accounts for another 6.8%; both are Montgomery-multiplication dominated. By Amdahl-style decomposition of the profiler symbol attribution, Montgomery multiplication accounts for approximately 53% of end-to-end proving time in this configuration; this is prevalence evidence, not a measured full-proof VAXEN speedup.

D. ML-DSA Signature Generation

Table VII reports ML-DSA signing, where the proxy implementation is derived from the Dilithium repository [10]. The projected 1.35–$1.54\times$ signing speedup suggests that the primitive-level benefit survives attenuation by hashing, sam-

TABLE VII
PROJECTED PERFORMANCE UNDER THE BALANCED PISA MAPPING.

Kernel	ISA	Rocket	Lion Cove	Skymont	Zen 5
BabyBear Mont. [20]	AVX2	7.90/4.80/1.65	9.38/6.85/1.37	14.22/7.66/1.86	4.59/2.79/1.65
BabyBear Mont. [21]	AVX-512	7.05/4.17/1.69	–	–	3.84/2.17/1.77
ML-DSA Mont. [10]	AVX2	5.60/3.83/1.46	6.70/5.63/1.19	11.89/5.55/2.14	3.07/1.94/1.58
ML-DSA NTT [10]	AVX2	920/588/1.56	1130/666/1.70	2189/1187/1.84	540/324/1.67
ML-DSA INTT [10]	AVX2	804/480/1.68	940/571/1.65	2193/1177/1.86	432/270/1.60
Poseidon2 $w = 16$ [22]	AVX2	628/421/1.49	672.49/517.39/1.30	1433.15/948.48/1.51	334/247/1.35
Poseidon2 $w = 16$ [23]	AVX-512	383/269/1.42	–	–	168/113/1.49
Poseidon2 $w = 24$ [22]	AVX2	1075/757/1.42	1089.49/836.19/1.30	2401.00/1682.01/1.43	527/391/1.35
Poseidon2 $w = 24$ [23]	AVX-512	656/487/1.35	–	–	279/199/1.40
BabyBear NTT-19	AVX2	67.97/45.55/1.49	94.12/63.22/1.49	124.71/80.37/1.55	41.88/28.61/1.46
BabyBear NTT-20	AVX2	69.37/46.67/1.49	106.51/75.47/1.41	133.42/88.47/1.51	42.93/29.74/1.44
Plonky3 NTT-19 [24]	AVX2	748.54/658.05/1.14	1081.64/993.69/1.09	1386.75/1125.55/1.23	391.86/344.49/1.14
Plonky3 NTT-19 [24]	AVX-512	642.47/604.09/1.06	–	–	323.48/312.43/1.04
Plonky3 NTT-20 [24]	AVX2	778.04/681.81/1.14	1129.41/1023.22/1.10	1473.37/1149.68/1.28	393.97/346.78/1.14
Plonky3 NTT-20 [24]	AVX-512	664.73/627.00/1.06	–	–	332.31/319.49/1.04
ML-DSA Sign [10]	AVX2	395785/292462/1.35	469204/336842/1.39	787517/512795/1.54	218403/160623/1.36
Philox [25]	AVX2	27/11/2.45	32/15/2.13	56/31/1.81	16/8/2.00
Philox [25]	AVX-512	17/6/2.83	–	–	8/3/2.67

Note: Cells report Base/VAXEN/speedup using median cycles. Poseidon2 is cycles per permutation; BabyBear NTT rows are cycles per scalar field element; NTT-19 denotes $\log n = 19$. Plonky3 NTT is the Rust dft_batch benchmark with width 32, normalized by transform length. Philox denotes Philox 4x32x10: four 32-bit counter words and 10 rounds. ML-DSA signing fixes the rejection-loop count to 5. Lion Cove/Skymont are AVX2-only; "–" marks unavailable AVX-512. VAXEN cells are proxy timings under Table IV.

pling, and the Fiat–Shamir rejection loop, which VAXEN does not accelerate.

E. Philox PRNG

Our Philox implementation is derived from the random123 library [25]. Table VII (lower rows) shows that the same primitive also benefits a non-cryptographic kernel. Philox is a direct consumer of 32-bit multiply-high, improving by $1.81\times$–$2.45\times$ on AVX2 and up to $2.83\times$ where AVX-512 is available.

VI. STATIC MICROARCHITECTURAL ANALYSIS

We complement the PISA evaluation with static scheduling analysis using llvm-mca [19] on Rocket Lake and Zen 5; we do not add Lion Cove/Skymont static-analysis rows because LLVM support for these scheduling models is still incomplete relative to our hardware measurements. We extract the timed hot paths from the benchmark build products and feed those snippets directly to the tool. We center the discussion on BabyBear Montgomery multiplication, ML-DSA Montgomery multiplication, and the ML-DSA NTT butterfly. These three kernels are chosen because they are the compute-bound components most directly affected by the missing 32-bit multiply-high instruction, and their straight-line code structure largely removes memory-access effects, making the port-pressure reduction from vpmulhd/vpmulhud clearer than in larger benchmark harnesses that also include memory traffic, MDS layers, or rejection-loop overhead.

A. Montgomery Multiplication

The upper rows of Table VIII summarize the BabyBear results, followed by the AVX2 ML-DSA Montgomery multiplication. Take BabyBear as an example: on AVX-512, Rocket Lake drops from 13 instructions and 5.0-cycle throughput to 7 instructions and 2.0 cycles ($2.50\times$ throughput speedup). The AVX2 ML-DSA Montgomery kernel shows the same mechanism, shrinking from 12 instructions to 4 and improving reciprocal throughput from 3.0 to 1.5 cycles on both Rocket Lake and Zen 5, while also shortening the critical path from 16 to 14 cycles on Rocket Lake and from 12 to 10 cycles on Zen 5.

B. ML-DSA NTT Butterfly

For ML-DSA, we analyze a representative butterfly core extracted from the AVX2 NTT-256 implementation [10]. The captured snippet contains four consecutive AVX2 Cooley–Tukey butterflies, each comprising a Montgomery multiplication and an add/subtract pair, which together cover 32 lane-level butterfly units. In contrast to the full ML-DSA NTT performance reported in Section V, which includes the overhead of memory accesses and coefficients permutations, this

TABLE VIII
STATIC-ANALYSIS SUMMARY FROM LLVM-MCA.

Kernel	ISA	Var.	Rocket				Zen 5			
			Inst	RT	SU	CP	Inst	RT	SU	CP
BabyBear Mont. [20]	AVX2	Base	14	3.5	1.00	23	14	3.0	1.00	17
		VAXEN	7	2.5	1.40	21	7	2.0	1.50	15
BabyBear Mont. [21]	AVX-512	Base	13	5.0	1.00	26	13	6.0	1.00	16
		VAXEN	7	2.0	2.50	21	7	4.0	1.50	15
ML-DSA Mont. [10]	AVX2	Base	12	3.0	1.00	16	12	3.0	1.00	12
		VAXEN	4	1.5	2.00	14	4	1.5	2.00	10
ML-DSA NTT [10]	AVX2	Base	60	12.0	1.00	30	60	12.0	1.00	25
		VAXEN	28	6.0	2.00	17	28	6.0	2.00	13

Note: Inst is instruction count, RT is block reciprocal throughput, SU is throughput speedup over the same-ISA baseline, and CP is critical-path length.

analysis focuses exclusively on the core butterfly computation instructions. By isolating these arithmetic primitives, we can more accurately evaluate the direct impact of the proposed VAXEN on the computational bottleneck.

Table VIII (lower rows) shows the results: the baseline uses 60 instructions per four-butterfly block with 12.0-cycle reciprocal throughput on both platforms, while the VAXEN version uses 28 instructions with 6.0-cycle throughput ($2.00\times$). The critical path also shrinks materially, from 30 to 17 cycles on Rocket Lake and from 25 to 13 cycles on Zen 5.

Figure 2a and Figure 2b show the per-port pressure profiles on Rocket Lake and Zen 5, respectively. On both platforms, the VAXEN variant substantially reduces sustained pressure on the dominant multiply-capable resources, which is consistent with the throughput gains in Table VIII.

VII. FORMAL VERIFICATION AND ASIC COST

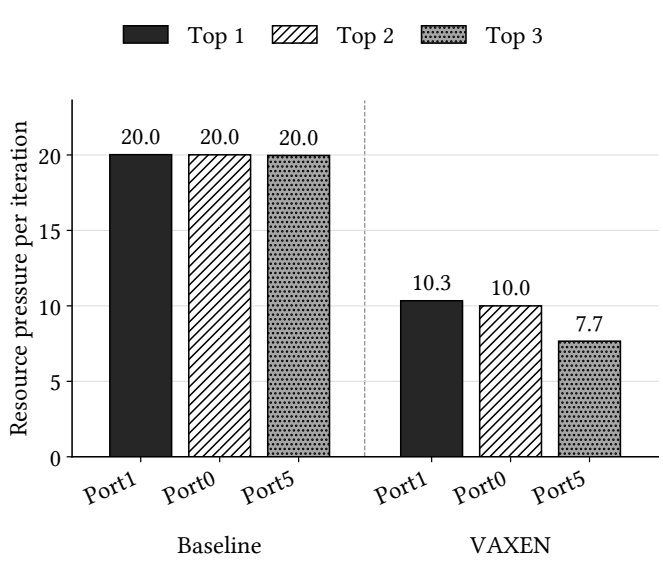
VAXEN is a minimal lane-wise primitive, so two complementary questions decide whether the proposal is credible: do the optimized proxy formulations preserve the intended semantics, and does the required datapath remain small relative to a contemporary x86 core? We answer the first with CRYPTOLINE [11] and the second with an ASIC evaluation library that instantiates the 16/32-bit AVX2 integer execution hardware in two configurations.

A. CRYPTOLINE *Validation*

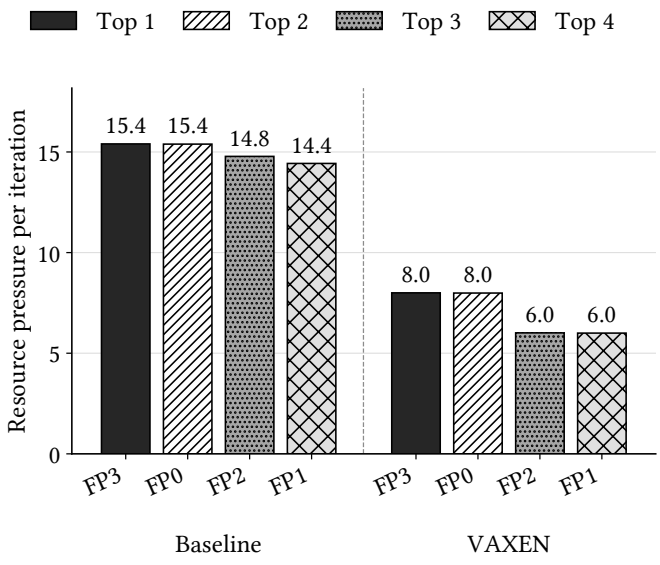
CRYPTOLINE [11] is a domain-specific language and verifier for low-level cryptographic arithmetic. A program is written as a straight-line sequence of fixed-width bit-vector operations annotated with a pre-condition and a post-condition that combine a range predicate and an algebraic equation. The tool lowers the program to SSA (Static Single Assignment) form, discharges range obligations with an SMT (Satisfiability Modulo Theories) solver, and discharges the algebraic equation symbolically modulo the declared invariants. A `PASS` result is a machine-checked proof that every execution respecting the pre-condition produces an output satisfying both predicates.

We encode the Montgomery multiplication routines that the rewritten kernels execute on a VAXEN-capable CPU: the BabyBear AVX2/AVX-512 routine built on vpmulhud (unsigned, $q = 2^{31} - 2^{27} + 1$) and the ML-DSA AVX2/AVX-512 routine built on vpmulhd (signed, $q = 8380417$). We also validate the Philox multiply-high kernel with the same framework. Each proof is a word-level model of the routine under the semantics VAXEN ascribes to the multiply-high instructions. For the two Montgomery routines the post-condition asserts both the arithmetic relation (the Montgomery product modulo q) on the vector routine's own outputs and the output-range bound that the surrounding software relies on. For the Philox kernel, transcribed with its vpmulhud/vpmulld high/low decomposition, the post-condition asserts word-level equivalence to an independent reference round function; a Philox output word carries no range bound because it spans the full 32-bit range by design. The obligations are discharged with per-instruction overflow checking disabled, as the reductions deliberately rely on well-defined modular wraparound, while the declared range and algebraic predicates are machine-checked.

These proofs pass. Consequently, under the semantics that VAXEN ascribes to vpmulhd/vpmulhud, the validated proxy routines preserve the reference arithmetic relation and the required input-output range contract. For BabyBear and ML-DSA, this means the AVX2/AVX-512 Montgomery multiplication routines remain mathematically equivalent to their reference formulations after the VAXEN rewrite. Because the ML-DSA NTT and full ML-DSA signing path reuse that same verified vpmulhd Montgomery multiplication routine, they inherit an indirect correctness guarantee along the software path that calls it. Table IX summarizes the three assurance levels used in this paper: direct CRYPTOLINE proofs, indirect coverage through a verified subroutine, and manual review for the remaining workloads.



(a) Intel Rocket Lake execution port pressure.



(b) AMD Zen 5 execution port pressure.

Fig. 2. ML-DSA AVX2 NTT butterfly static analysis using llvm-mca.

TABLE IX
ASSURANCE LEVELS ACROSS THE VALIDATED WORKLOADS.

Workload	Level	Guarantee
BabyBear Mont.	Direct	Algebraic + range
ML-DSA Mont.	Direct	Algebraic + range
Philox	Direct	Bit-level mulhi
ML-DSA NTT/Sign	Indirect	Via ML-DSA Mont.
Others	Manual	Code review only

Note: Direct rows are machine-checked with CRYPTOLINE. "Indirect" reuses the verified ML-DSA Montgomery routine under the same software path. "Manual" means code review rather than a machine-checked proof.

B. ASIC Evaluation Library

The baseline cluster implements the 16/32-bit AVX2 integer execution hardware that Zen 4 and Zen 5 pipe 0–3 must support: add/sub, logical, compare, min/max, blend/shuffle, shift, and the full 16/32-bit multiply family. Our VAXEN configuration adds vpmulhd and vpmulhud; every other resource is held fixed.

Pipeline and decode. Both configurations share the same four-pipe cluster, pipe issue rules, and per-instruction latency/CPI contract as Zen 4 and Zen 5 (vpmulld/vpmuludq at latency 3, CPI 0.5 on pipe 0/1; the extended instructions inherit the same binding). Instruction decode is a combinational one-hot predecode on the 6-bit opcode feeding per-group input registers; adding the two new opcodes touches only the predecode table and the VectorMul32x8 result multiplexer (MUX), so the front end is not a source of marginal area.

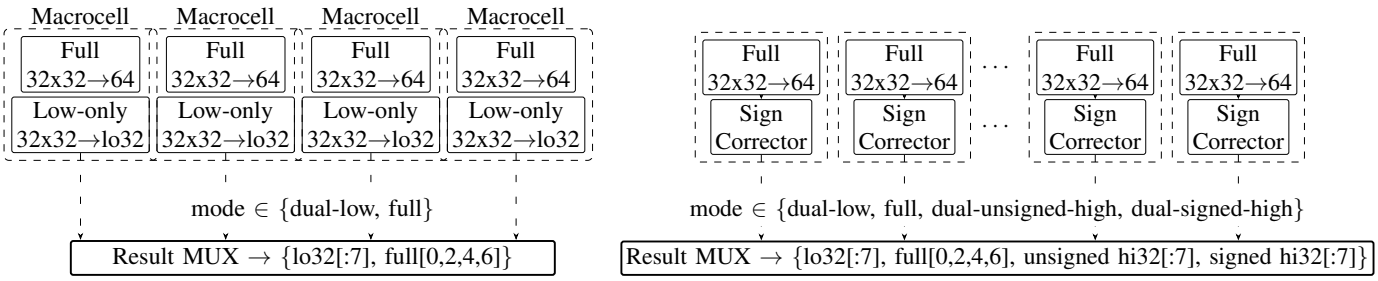
Toolchain. The datapath is written in Chisel 3 and elaborated to SystemVerilog through FIRRTL. Logic synthesis uses Yosys against the open ASAP7 7 nm standard-cell library [26], followed by a fanout-driven buffer insertion pass that repairs Yosys's min-size cell mapping. Static timing is reported by OpenSTA against the same library at a 1 GHz target. Functional correctness is checked through ChiselTest regressions that cover the ALU, shift, 16-bit multiply, and both 32-bit multiply configurations against a bit-accurate golden model.

C. Baseline and VAXEN Microarchitecture

Figure 3 illustrates the VectorMul32x8 architecture for the two configurations. The baseline uses 4 dual-mode macrocells per 256-bit multiply slice: each macrocell pairs a FullMul32Lane with a LowOnlyMul32Lane. In full mode, i.e., when a vpmuldq or vpmuludq instruction is executed, only the FullMul32Lane is active and produces the 64-bit even-lane product; in dual-low mode, i.e., when the vpmulld instruction is executed, the FullMul32Lane's low 32 bits serve the even 32-bit lanes while the LowOnlyMul32Lane serves the odd 32-bit lanes.

FullMul32Lane and LowOnlyMul32Lane share the same Booth radix-4 encoding, partial-product generator, and Wallace tree front end. The only structural difference is Stage 3: a 64-bit prefix adder on the full lane and a 32-bit prefix adder on the low-only lane. This pairing exactly matches the ISA asymmetry exposed by the existing AVX2 32-bit multiply family and avoids paying full-product cost on odd lanes.

Our VAXEN configuration replaces the four dual-mode macrocells with eight FullMul32Lane instances, so every 32-bit lane produces a full 64-bit product. An 8-wide MulHiSignedCorrector is used to convert the unsigned high half into signed for the vpmulhd instruction. The Stage-3 output MUX of baseline is widened with two cases for vpmulhud and vpmulhd. All three existing 32-bit instructions (vpmulld,



(a) Baseline: VectorMul32x8 for vpmulld/vpmuludq (pipe 0/1), implemented as 4 dual-mode macrocells (4 full + 4 low-only lanes) per 256-bit slice.

(b) VAXEN: VectorMul32x8 for vpmulld/vpmuludq/vpmulhd/vpmulhud (pipe 0/1), implemented with 8 full 32x32→64 lanes plus per-lane signed high-half correctors.

Fig. 3. VectorMul32x8 execution-unit organization. Both configurations share the same 3-stage Booth+Wallace+prefix-adder pipeline and the same latency/CPI contract.

TABLE X
POST-SYNTHESIS CLUSTER AREA ON ASAP7 7 NM AT 1 GHZ.

Component	Base	VAXEN	Δ
4 × ALU predecode	4405.4	4405.4	0
2 × 16x16 mul	4999.4	4999.4	0
2 × 32x8 mul	7662.3	8478.6	+816.3
Wrapper	1064.1	1064.1	0
Cluster total	18131.2	18947.5	+816.3

Note: Area values are in μm^2 and include sub-hierarchy. Only the 32x8 multiplier block differs between baseline and VAXEN.

vpmuldq, and vpmuludq) preserve their original datapath and timing.

D. Marginal Cost

Both configurations meet the 1 GHz target with equal worst reg-to-reg arrival (908 ps), and the critical path is the same Wallace-tree plus 64-bit prefix-adder chain inside FullMul32Lane. Adding vpmulhd/vpmulhud therefore does not shift the frequency envelope. Table X reports the post-synthesis area decomposition. The entire increment is localized to VectorMul32x8: the ALU cluster, 16-bit multiplier, and shift fabric are bit-identical between the two builds.

The absolute increment is $A_{\text{ext}} - A_{\text{base}} = 816.3\,\mu m^2 = 8.16 \times 10^{-4}\,mm^2$. Normalized against the baseline cluster, $\Delta_{\text{base}} = +4.50\%$. This relative number is a useful sanity check inside the same design boundary. A more realistic scale is the area of a single modern x86 core:

- Against an AMD Zen 4 core ($\approx 3.84\,mm^2$ including L2 cache [27]), the VAXEN increment represents $\Delta_{\text{Zen4}} \approx 0.021\%$.
- Against an AMD Zen 5 core ($\approx 4.77\,mm^2$ including L2 cache [28]), the increment represents $\Delta_{\text{Zen5}} \approx 0.017\%$.

The dominant source of the increment is the move from 4 FullMul32Lanes + 4 LowOnlyMul32Lanes to 8 FullMul32Lanes per slice: specifically, each low-only lane's 32-bit Stage-3 prefix adder is replaced by a 64-bit prefix adder, while the shared Booth–Wallace front end is reused.

Post-place-and-route validation. To validate the post-synthesis results, we run both configurations through the OpenROAD-flow-scripts (ORFS) backend targeting 2.5 GHz on the same ASAP7 library. Both configurations achieve post-route $F_{\text{max}} > 1$ GHz (baseline: 1112.5 MHz; VAXEN: 1086.8 MHz), confirming that the extension keeps the design above the 1 GHz target. The post-route design area increases by $+3328.3\,\mu m^2$ ($+14.33\%$ of the baseline cluster), and total power increases by $+0.084$ W ($+18.7\%$ of the baseline cluster). Normalized against a Zen 5 core, the post-route increment is $\approx 0.070\%$, still well below 0.1%. The gap between the post-synthesis ($+4.50\%$) and post-route ($+14.33\%$) increments is traced to ORFS's internal ABC speed-oriented technology mapping, which produces a structurally different cell selection for larger netlists regardless of the clock constraint. We therefore report the Yosys post-synthesis numbers as the primary result reflecting the true RTL-level structural increment, and the ORFS post-route numbers as a conservative upper bound that accounts for backend physical-design effects.

E. Alternative Implementation Strategy

A less invasive option is a *MUX-only* approach: leave the baseline 4 FullMul32Lanes + 4 LowOnlyMul32Lanes structure untouched and add only a result MUX to read the upper 32 bits from the existing even-lane full multipliers. Because the odd-lane LowOnlyMul32Lane units cannot produce the high half, the instruction must be decomposed into two micro-operations that time-share the full lanes, yielding CPI ≈ 1.0 instead of 0.5. This near-zero-area point is the MUX-only lower bound in Table VI. Our full design upgrades the odd-lane prefix adders from 32-bit to 64-bit, costing $+0.017\%$ of a Zen 5 core while preserving the balanced CPI 0.5 contract.

VIII. DISCUSSION

The evaluation is deliberately single-core to isolate the ISA-level contribution. This section discusses broader implications.

A. Multi-Core Throughput Outlook

Improving TLS server throughput by accelerating single-core cryptographic primitives has been studied extensively [31], [32]. ML-DSA signing is a core server-side

VAXEN MontMul on AVX2 (Section III)

```
ml = vpmulld(a, bqi)
ch = vpmulhd(a, b)
th = vpmulhd(ml, q)
r  = vpsubd(ch, th)
```

Barrett Multiplication on NEON [29]

```
z = mul(a, b)
t = sqrdmulh(a, bbar)
r = mls(z, t, q)
```

MontMul on RVV [30]

```
m  = vmul.vx(a, bqi)
ch = vmulh.vx(a, b)
th = vmulh.vx(m, q)
r  = vsub.vv(ch, th)
```

Fig. 4. Modular-multiplication kernels in ML-DSA NTT when one multiplicand is known and can be precomputed.

operation in post-quantum TLS handshakes [16], [17], and throughput-oriented servers naturally exploit multiple cores via worker threads. Per-core gains from VAXEN should scale to the multi-core setting as long as shared memory and cache subsystems do not become the new bottleneck. We mark this as a deployment outlook rather than a measured claim.

B. Portability of the Architectural Argument

The missing 32-bit lane-wise multiply-high is structural, not a compiler scheduling artifact. On x86, the same emulation sequence has been the canonical path since Skylake (2015) on Intel and Zen 3 (2020) on AMD because the underlying vpmuludq/vpmulld contract has been stable across generations [14]; no reachable compiler rewrite removes the widening-plus-repack shape. Other SIMD ecosystems make different tradeoffs at the same point in the design space. ARM NEON exposes doubling multiply-high instructions such as sqrdmulh, which have been shown to accelerate modular multiplication effectively in optimized Neon NTT implementations [29]; RISC-V RVV exposes vmulhu and vmulh at arbitrary element widths under its length-agnostic model. VAXEN closes an existing gap in x86 SIMD rather than proposing a new cross-ISA capability, and its portability argument is that closely related arithmetic support already has a natural form in neighboring ISAs.

The realized benefit is microarchitecture-sensitive: AMD already exhibits balanced 16/32-bit multiply performance, while Intel's vpmulld remains slower than the 16-bit family. VAXEN closes the same ISA-level gap in both cases; the magnitude of the gain depends on whether the implementation follows an AMD-like balanced path or a slower Intel-like decomposition (Section IV).

C. Software Adoption Path

The target libraries already use hand-written intrinsics or assembly for hot SIMD kernels, so VAXEN can be adopted first through intrinsics and small library macros. Our 198-line LLVM 20 prototype exposes vpmulhd/vpmulhud intrinsics and backend definitions; it is not a production compiler integration, but it shows a direct path once the ISA exists. Clang 20 already recognizes the portable multiply-high idiom as `ISD::MULHU`: the 16-bit form lowers to vpmulhuw, while the 32-bit form falls back to widening-plus-repack code (10 vpmuludq, 15 vpshufd, 5 vpblendd) because AVX lacks a matching instruction. With vpmulhd/vpmulhud patterns, the same recognition lowers the 32-bit idiom to one instruction; on current AVX, software-only rewrites still require widening-plus-repack.

D. Relationship to AVX10 and the x86 ISA Roadmap

Intel's AVX10 initiative reorganizes AVX-512 subextensions into a single versioned feature set. AVX10.1 [18] introduces no new instructions, and AVX10.2 [33] focuses on floating-point and AI data types. As of April 2026, neither AVX10.1 nor AVX10.2 addresses the 32-bit multiply-high gap, and no publicly documented Intel or AMD ISA extension includes vpmulhd or vpmulhud.

Intel's discontinued Knights Corner did include 32-bit multiply-high in its proprietary SIMD ISA, but under an encoding incompatible with VEX/EVEX that was never carried into AVX-512 or AVX10; the precedent shows the primitive has been considered useful before.

VAXEN is designed to be forward-compatible with the AVX10 trajectory: the proposed instructions use the standard VEX/EVEX encoding within the existing 66.0F38 opcode map (Section III), and a dedicated CPUID feature bit gates their availability. If a future AVX10 revision adopts 32-bit multiply-high, the VAXEN encoding offers a concrete starting point; a different opcode assignment would not affect our software-level analysis.

E. Scope and Limitations

The central claim is narrow: a lane-wise 32-bit multiply-high primitive removes a recurring bottleneck in BabyBear and ML-DSA Montgomery multiplication, ML-DSA NTT and signature generation, the Montgomery-heavy portion of Plonky3 Poseidon2, and Philox PRNG, at low marginal core-area overhead. Two related cryptographic workloads are intentionally excluded. ML-KEM [13] uses a 16-bit modulus ($q = 3329$), so its NTT already benefits from the existing vpmulhw and is not affected by the 32-bit gap. Homomorphic encryption (HE) schemes such as BFV and BGV typically operate on 50–60 bit prime moduli per level of a residue number system (RNS) chain [34], [35], making them consumers of multi-word arithmetic rather than the lane-local 32-bit multiply-high targeted by VAXEN. The main performance tables (e.g., Table VII) should be read as AMD-like balanced-multiply projections, with the Intel-like slow-vpmulld mapping serving only as a conservative sensitivity point. The formal verification evidence is confined to the validated proxy routines and the indirect reuse paths described in Section VII; it does not extend to every surrounding wrapper or benchmark harness.

F. Future Directions

A natural extension is to ask whether the same minimal-patch philosophy applies to matrix engines such as tensor cores and NPUs [36], [37], which currently lack finite-field arithmetic support.

IX. Related Work

VAXEN sits at the intersection of three lines of prior work: highly tuned software for PQC arithmetic on existing SIMD ISAs, ISA or accelerator proposals for cryptographic kernels, and cross-architecture studies showing how arithmetic support reshapes the implementation space.

A. Optimized Software for PQC

On x86, high-performance NTT and modular-arithmetic implementations have long depended on ISA-specific reduction formulas and packing choices rather than direct support for all desired arithmetic primitives. Seiler [12] showed that AVX2 integer NTTs (targeting moduli $q < 2^{16}$, e.g., $q = 3329$ for ML-KEM [13], [38] and $q = 12289$ for NewHope [39]) can substantially outperform prior floating-point formulations by combining dense packing with a modified Montgomery reduction tailored to the available instructions.

The same pattern appears on other SIMD ecosystems. Neon NTT utilizes Barrett [40] multiplication with Neon-specific vector idioms and, notably, uses doubling multiply-high instructions (e.g., sqrdmulh) to accelerate modular multiplication effectively on Armv8-A [29]. On RISC-V, recent optimized software for scalar and RVV-enabled variants shows that different ISAs lead to materially different choices in reduction strategy, scheduling, and vectorization style [30].

VAXEN complements this line of work rather than replacing it: instead of proposing another software workaround, it asks whether one recurring x86 SIMD 32-bit multiply-high gap should become a first-class SIMD primitive.

B. ISA Extensions

Our closest architectural reference point is MQX [8], a MICRO 2025 AVX-512 extension for multi-word cryptographic arithmetic evaluated through PISA. VAXEN is narrower: MQX adds carry-aware 64-bit word operations for large-integer kernels, whereas VAXEN targets the lane-local 32-bit multiply-high bottleneck in PQC, ZKP, and Philox PRNG. Table III quantifies this complementarity for ML-DSA Montgomery multiplication. The overlap is methodological and architectural rather than semantic. We do not claim that multiply-high, PISA, or CryptoLine are new. The contribution is to identify a long-standing x86 AVX gap, show that it is now a shared bottleneck across three domains, and show through measured proxy isolation (Table VI) and synthesis that the gap can be closed at low relative hardware cost.

C. Cross-Architecture Perspective

Figure 4 makes cross-ISA comparison concrete by placing ML-DSA NTT modular-multiplication kernels from VAXEN-enabled AVX2 MontMul, NeonNTT-style NEON Barrett multiplication, and RVV MontMul side by side.

X. Conclusion

x86 SIMD still lacks a direct 32-bit multiply-high primitive even though that operation appears repeatedly in modern ZKP systems, PQC, and Philox PRNG. This paper argues that the gap is both practically important and architecturally fixable.

VAXEN addresses this problem with a minimal AVX2 and AVX-512 extension that adds signed and unsigned 32-bit multiply-high (i.e., vpmulhd and vpmulhud) with per-lane semantics. Using PISA, static analysis, CryptoLine validation, ASIC cost estimation, and a discussion of practical deployment, the paper builds a multi-angle case that the proposed primitive is versatile, useful, and inexpensive enough to merit serious consideration.

The broader takeaway is that narrow arithmetic primitives can have outsized impact when they sit on critical paths shared across multiple domains. For x86 SIMD, our proposed 32-bit multiply-high instructions are such a primitive.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (Grant No. 62502275).

References

[1] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," *IACR Cryptology ePrint Archive*, vol. 2019, p. 953, 2019. [Online]. Available: https://eprint.iacr.org/2019/953

[2] Plonky3 Contributors, "Plonky3," GitHub repository, 2023, accessed 2026-04-01. [Online]. Available: https://github.com/Plonky3/Plonky3

[3] National Institute of Standards and Technology, "Module-Lattice-Based Digital Signature Standard," National Institute of Standards and Technology, Federal Information Processing Standards Publication 204, Aug. 2024. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.204

[4] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw, "Parallel random numbers: As easy as 1, 2, 3," in *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*. New York, NY, USA: ACM, 2011, pp. 16:1–16:12. [Online]. Available: https://doi.org/10.1145/2063384.2063405

[5] PyTorch Contributors, "Philoxrngengine.h," PyTorch source file, 2025, accessed 2026-04-01. [Online]. Available: https://github.com/pytorch/pytorch/blob/main/aten/src/ATen/core/PhiloxRNGEngine.h

[6] TensorFlow Authors, "tf.random.generator," TensorFlow API documentation, 2024, documents Philox support; accessed 2026-04-01. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/random/Generator

[7] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985. [Online]. Available: https://doi.org/10.2307/2007970

[8] N. Zhang, S. Fu, and F. Franchetti, "Towards closing the performance gap for cryptographic kernels between cpus and specialized hardware," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture, MICRO 2025, Seoul, Republic of Korea, October 18-22, 2025*. New York, NY, USA: ACM, 2025, pp. 1704–1718. [Online]. Available: https://doi.org/10.1145/3725843.3756120

[9] L. Grassi, D. Khovratovich, and M. Schofnegger, "Poseidon2: A faster version of the poseidon hash function," in *Progress in Cryptology - AFRICACRYPT 2023 - 14th International Conference on Cryptology in Africa, Sousse, Tunisia, July 19-21, 2023, Proceedings*, ser. Lecture Notes in Computer Science, N. E. Mrabet, L. D. Feo, and S. Duquesne, Eds. Cham, Switzerland: Springer, 2023, pp. 177–203. [Online]. Available: https://doi.org/10.1007/978-3-031-37679-5_8

[10] The PQ-Crystals Team, "CRYSTALS-Dilithium AVX2 implementation," https://github.com/pq-crystals/dilithium/tree/master/avx2, 2024, accessed: 2026-04-01. Specifically `ntt.S` and `invntt.S` routines.

[11] Y.-F. Fu, J. Liu, X. Shi, M.-H. Tsai, B.-Y. Wang, and B.-Y. Yang, "Signed cryptographic program verification with typed cryptoline," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. New York, NY, USA: ACM, 2019, pp. 1591–1606. [Online]. Available: https://doi.org/10.1145/3319535.3354199

[12] G. Seiler, "Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography," *IACR Cryptology ePrint Archive*, vol. 2018, p. 39, 2018. [Online]. Available: https://eprint.iacr.org/2018/039

[13] National Institute of Standards and Technology, "Module-Lattice-Based Key-Encapsulation Mechanism Standard," National Institute of Standards and Technology, Federal Information Processing Standards Publication 203, Aug. 2024. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.203

[14] A. Abel and J. Reineke, "uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures," in *ASPLOS*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 673–686, the website https://uops.info/table.html was accessed on 2026-04-01. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304062

[15] 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.

[16] D. Sikeridis, P. Kampanakis, and M. Devetsikiotis, "Post-quantum authentication in TLS 1.3: A performance study," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/post-quantum-authentication-in-tls-1-3-a-performance-study/

[17] R. Shekh-Yusef, H. Tschofenig, M. Ounsworth, T. Reddy.K, and Y. Rosomakho, "Post-Quantum Traditional (PQ/T) Hybrid Authentication with Dual Certificates in TLS 1.3," Internet Engineering Task Force, Internet-Draft draft-yusef-tls-pqt-dual-certs-01, Dec. 2025, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-yusef-tls-pqt-dual-certs/01/

[18] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, Intel Corporation, Dec. 2024, accessed: 2026-04-01. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[19] LLVM Project, *llvm-mca - LLVM Machine Code Analyzer*, LLVM Foundation, 2026, accessed: 2026-04-01. [Online]. Available: https://llvm.org/docs/CommandGuide/llvm-mca.html

[20] Plonky3 Contributors, "Plonky3: AVX2 implementation of BabyBear Montgomery multiplication," https://github.com/Plonky3/Plonky3/blob/main/monty-31/src/x86_64_avx2/packing.rs, 2023, accessed: 2026-04-01. Specifically the `impl Mul` for `PackedMontyParameters`.

[21] ——, "Plonky3: AVX-512 implementation of BabyBear Montgomery multiplication," https://github.com/Plonky3/Plonky3/blob/main/monty-31/src/x86_64_avx512/packing.rs, 2023, accessed: 2026-04-01. Specifically the `impl Mul` for `PackedMontyParameters`.

[22] ——, "Plonky3: AVX2 implementation of Poseidon2," https://github.com/Plonky3/Plonky3/blob/main/monty-31/src/x86_64_avx2/poseidon2.rs, 2023, accessed: 2026-04-01.

[23] ——, "Plonky3: AVX-512 implementation of Poseidon2," https://github.com/Plonky3/Plonky3/blob/main/monty-31/src/x86_64_avx512/poseidon2.rs, 2023, accessed: 2026-04-01.

[24] ——, "Plonky3: RecursiveDft implementation," https://github.com/Plonky3/Plonky3/blob/main/monty-31/src/dft/mod.rs, 2023, accessed: 2026-04-01. Specifically the `RecursiveDft` implementation.

[25] D. E. Shaw, "The random123 library of counter-based random number generators," https://github.com/DEShawResearch/random123/blob/main/include/Random123/philox.h, 2022, specifically the `philox.h` header implementation. Accessed: 2026-04-01.

[26] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "Asap7: A 7-nm finfet predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.

[27] H. Mujtaba, "AMD Discloses Zen 4C Architecture Details: Same Design As Zen 4, 35% Smaller, 2x Density & Cores," https://perma.cc/RV6E-6L3A, 2023, accessed: 2026-04-01.

[28] A. Garreffa, "Check out these beautiful high-res die shots of AMD's new Zen 5 CCD, ready for 3D V-Cache CPUs," https://perma.cc/NB5G-Z3HU, 2024, accessed: 2026-04-08.

[29] H. Becker, V. Hwang, M. J. Kannwischer, B.-Y. Yang, and S.-Y. Yang, "Neon NTT: Faster dilithium, kyber, and saber on cortex-A72 and apple M1," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 1, pp. 221–244, 2022. [Online]. Available: https://doi.org/10.46586/tches.v2022.i1.221-244

[30] J. Zhang, Y. Yan, J. Huang, and Ç. K. Koç, "Optimized software implementation of keccak, kyber, and dilithium on RV{32,64}IM{B}{V}," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2025, no. 1, pp. 632–655, 2025. [Online]. Available: https://doi.org/10.46586/tches.v2025.i1.632-655

[31] J. Zhang, J. Huang, L. Zhao, D. Chen, and Ç. K. Koç, "ENG25519: faster TLS 1.3 handshake using optimized X25519 and ed25519," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-jipeng

[32] D. J. Bernstein, B. B. Brumley, M. Chen, and N. Tuveri, "Opensslntru: Faster post-quantum TLS key exchange," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 845–862. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/bernstein

[33] Intel Corporation, *Intel® Advanced Vector Extensions 10.2 (Intel® AVX10.2) Architecture Specification*, Intel Corporation, Feb. 2026, revision 6.0. Accessed: 2026-04-08. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/828965/intel-advanced-vector-extensions-10-2-intel-avx10-2-architecture-specification.html

[34] F. Boemer, S. Kim, G. Seifu, F. D. M. de Souza, and V. Gopal, "Intel HEXL: Accelerating homomorphic encryption with intel AVX512-IFMA52," in *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. New York, NY, USA: ACM, 2021, pp. 57–62. [Online]. Available: https://doi.org/10.1145/3474366.3486926

[35] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012. [Online]. Available: https://eprint.iacr.org/2012/144

[36] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA tensor core programmability, performance & precision," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2018, Vancouver, BC, Canada, May 21-25, 2018*. Piscataway, NJ, USA: IEEE, 2018, pp. 522–531. [Online]. Available: https://doi.org/10.1109/IPDPSW.2018.00091

[37] Google Developers, "What is the edge TPU?" Coral documentation, 2020, documents Coral Edge TPU as a low-power inference ASIC; accessed 2026-04-01. [Online]. Available: https://coral.ai/docs/edgetpu/faq/

[38] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS - Kyber: A CCA-secure module-lattice-based KEM," in *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. Piscataway, NJ, USA: IEEE, 2018, pp. 353–367. [Online]. Available: https://doi.org/10.1109/EuroSP.2018.00032

[39] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange - A new hope," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds. USENIX Association, 2016, pp. 327–343. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim

[40] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*. Berlin, Heidelberg: Springer, 1986, pp. 311–323. [Online]. Available: https://doi.org/10.1007/3-540-47721-7_24

APPENDIX A
ARTIFACT APPENDIX

A. Abstract

This artifact reproduces the two core results of the paper: Table VII (projected performance under the balanced PISA mapping) and Table X (post-synthesis cluster area on ASAP7 7 nm). The software side is a unified `reproduce.sh` entrypoint that builds the cryptographic and non-cryptographic microbenchmarks and the upstream Plonky3 and Dilithium reproductions, then emits the projected-performance table. The hardware side is a self-contained Docker image that elaborates the baseline and VAXEN-extended execution clusters in Chisel

and drives them through Yosys synthesis and OpenROAD place-and-route on the open ASAP7 library. Together the two results support the paper's central claim that VAXEN delivers a consistent cross-kernel speedup at a negligible silicon cost. The artifact is publicly available at https://github.com/Ji-Peng/VAXEN_Artifact.

B. Artifact check-list (meta-information)

- **Program:** cryptographic and non-cryptographic SIMD microbenchmarks (BabyBear/ML-DSA Montgomery multiplication and NTT, Poseidon2, Philox) and a Chisel RTL evaluation library.
- **Compilation:** host GCC/Clang and a Rust toolchain for the benchmarks (installed automatically); Chisel/Yosys/OpenROAD are pinned inside the Docker image.
- **Run-time environment:** Ubuntu 20.04/22.04; Docker for the ASIC flow.
- **Hardware:** an AMD Zen 5 machine for the Zen 5 column of Table VII (e.g. a Google Cloud `c4d-standard-16`, an 8 physical core AMD EPYC 9B45); any x86-64 host with at least 32 GB RAM for the ASIC flow.
- **Metrics:** median cycle counts (Base/VAXEN/speedup) for Table VII; standard-cell area (μm^2) and worst reg-to-reg arrival for Table X.
- **Output:** `table7.md` (Table VII) and `asic_tables.md` (Table X).
- **Experiments:** one command per table (see below).
- **How much disk space required (approximately)?** A few GB, plus the Docker images.
- **How much time is needed to prepare workflow (approximately)?** About 30 minutes (dependency install and Docker image build).
- **How much time is needed to complete experiments (approximately)?** The Table VII run takes tens of minutes; the ASIC post-synthesis stage a few minutes, and the full place-and-route pass 3–4 hours.
- **Publicly available?** Yes.
- **Workflow automation framework used?** A shell entrypoint (`reproduce.sh`) plus Docker.

C. Description

1) How to access: Clone the public repository:

```
git clone \
  https://github.com/Ji-Peng/VAXEN_Artifact
cd VAXEN_Artifact
```

2) Hardware dependencies: The Zen 5 column of Table VII requires an AMD Zen 5 machine; we recommend a Google Cloud `c4d-standard-16` instance: an 8-core AMD EPYC 9B45 run with SMT disabled (8 vCPUs, one per physical core, ≈62 GB RAM), matching the paper's Zen 5 setup. If it helps, we can provide SSH access to a preconfigured instance on request. The ASIC flow (Table X) needs only an x86-64 host with at least 32 GB of RAM.

3) Software dependencies: Ubuntu 20.04/22.04. The performance flow installs its own dependencies (`build-essential`, `git`, `util-linux`, `ca-certificates`, `python3`, `curl`, and a Rust toolchain via `rustup`) when invoked with `--install-deps`. The ASIC flow requires only Docker; the image pins OpenROAD, Yosys, OpenSTA, and the open ASAP7 library, so no proprietary EDA tools are needed.

D. Installation

For the software results, no manual build step is required beyond dependency installation, which the entrypoint performs on first run. For the hardware results, build the Docker image once:

```
cd avx2-eval-lib
docker build -f docker/Dockerfile \
    -t avx2ext-asic .
```

E. Experiment workflow

Table VII (projected performance). On the Zen 5 machine, run:

```
./reproduce.sh pisa --install-deps \
    --output-root results/pisa
```

This produces `results/pisa/table7.md`.
Table X (ASIC area). From `avx2-eval-lib/` (the directory of the built image):

```
mkdir -p repro_out
docker run --rm \
    -v "$PWD/repro_out:/out" \
    avx2ext-asic all
```

The regenerated table lands at:

```
repro_out/asic_tables/asic_tables.md
```

Use `avx2ext-asic postsynth` to run only the fast area/timing stage, whose raw report holds the Table X area numbers.

F. Evaluation and expected results

The generated `table7.md` reproduces the Zen 5 column of Table VII: for each kernel it reports the baseline and VAXEN median cycle counts and their speedup. Cycle-level microbenchmarks vary run to run, so agreement within about 5% is expected. The ASIC flow reproduces the post-synthesis cluster area of Table X (18131.2 vs $18947.5\,\mu m^2$, a $+816.34\,\mu m^2$ or $+4.50\%$ increment) and the 908 ps worst reg-to-reg arrival to the decimal, using the pinned Yosys 0.63; the post-place-and-route figures are regenerated as a conservative upper bound and may vary by a few percent with the bundled OpenROAD version.

We scope this artifact-evaluation reproduction to the Zen 5 column of Table VII and to Table X, because Zen 5 is the hardware for which we can offer reviewers direct SSH access. The remaining Table VII columns (Rocket Lake, Lion Cove, Skymont) are reproduced by the identical command on the corresponding Intel machines; the artifact README documents that path for readers who have the hardware.

G. Methodology

Submission, reviewing, and badging followed the standard process:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae