Unified Hardware Architecture for Efficient Paillier Homomorphic Encryption

Fergus Xu
Department of Electrical and Computer Engineering
University of Washington
Seattle, WA, USA
xu000380@uw.edu

***Abstract*—This paper presents a hardware accelerator for the Paillier cryptosystem, enabling efficient additive homomorphic encryption and decryption in latency-sensitive applications. Paillier's partial homomorphic properties makes it attractive for secure aggregation, federated learning, and privacy-preserving analytics. However, the scheme's reliance on large modular exponentiations poses significant performance challenges in software. To address this, we design a parameterizable accelerator built around a systolic Montgomery multiplication array and a unified modular exponentiation datapath. Cycle-accurate simulation and functional verification confirm both correctness and predictable timing behavior. This work demonstrates a scalable, reusable hardware template for privacy-preserving computation under strict performance and security constraints.**

***Index Terms*—Homomorphic encryption, Paillier cryptosystem, Montgomery multiplication, hardware accelerator, modular exponentiation, systolic array**

I. Introduction

This project focuses on hardware acceleration for additive homomorphic encryption, with particular emphasis on the Paillier cryptosystem [1]. Paillier supports computations directly on ciphertexts—specifically, addition and scalar multiplication—making it well-suited for secure aggregation, federated learning, and privacy-preserving analytics.

However, these benefits come at the cost of heavy computation. Paillier encryption and decryption both rely on large-integer modular exponentiation, often involving 1024–4096 bit operands. This level of latency presents a barrier to real-time or embedded system use.

To address this, we present a cycle-deterministic, low-latency hardware accelerator optimized for Paillier's core arithmetic. The design features a shared datapath for encryption and decryption, built around a pipelined systolic Montgomery multiplier. By exploiting operand reuse and tightly timed scheduling, the system maintains high throughput across a range of key sizes. The implementation is also fully parameterized, enabling deployment with different limb counts and bitwidths without major redesign with consistent performance characteristics.

A. Hardware Acceleration?

Paillier's ability to perform encrypted-domain operations is ideal for modern privacy-preserving computing tasks—but its arithmetic complexity is a major bottleneck. General-purpose CPUs are not optimized for modular arithmetic, especially with large operand sizes. As a result, even highly optimized software libraries often fall short of latency requirements for interactive applications [2].

Hardware acceleration offers a clear path forward. By designing an architecture around fixed-latency modular multiplication and deterministic operand scheduling, we can dramatically improve throughput and reduce power consumption. In particular, our system uses the Coarsely Integrated Operand Scanning (CIOS) method for Montgomery multiplication, which offers a good tradeoff between hardware complexity and speed [3]. The design also ensures constant-time execution, reducing side-channel vulnerabilities by avoiding secret-dependent branching or timing behavior [4].

B. Exponentiation via Montgomery Ladder

Modular exponentiation is implemented using a square-and-multiply ladder algorithm [4], chosen for its uniform control path. The ladder ensures that each bit of the exponent triggers both multiplication and squaring operations, independent of its value. Although our current design does not eliminate all side-channel vectors, specifically, cache timing attacks, the ladder serves as a foundation for future secure variants [7].

II. Background

This section introduces the cryptographic foundations of our work, focusing on the Paillier cryptosystem and its homomorphic properties. We then discuss the computational challenges associated with implementing Paillier in hardware and motivate the need for a modular arithmetic accelerator. Finally, we review prior work in cryptographic hardware design, highlighting the design space our approach addresses.

A. Paillier Cryptosystem Overview

The Paillier cryptosystem [1] is a probabilistic public-key encryption scheme based on the Decisional Composite Residuosity Assumption (DCRA). It is notable for supporting additive homomorphism over ciphertexts, making it suitable for privacy-preserving aggregation tasks.

1) Key Generation: To generate the Paillier key pair:

1) Select two large, distinct prime numbers p and q.
2) Compute the modulus:

$$n = p \cdot q, \quad n^2 = n \cdot n$$

3) Compute Lambda:

$$\lambda = \text{lcm}(p-1, q-1)$$

4) Choose a generator $g \in \mathbb{Z}_{n^2}^*$ such that:

$$\gcd(L(g^\lambda \bmod n^2), n) = 1$$

where $L(u)$ is the Paillier L-function:

$$L(u) = \frac{u-1}{n}$$

5) Compute:

$$\mu = \left(L(g^\lambda \bmod n^2)\right)^{-1} \bmod n$$

The public key is (n, g), and the private key is (λ, μ).

2) Encryption: Given a plaintext message $m \in \mathbb{Z}_n$ and a randomly chosen $r \in \mathbb{Z}_n^*$, the ciphertext c is computed as:

$$c = g^m \cdot r^n \mod n^2$$

This encryption is probabilistic: the same message encrypted with different values of r yields different ciphertexts, providing semantic security.

3) Decryption: To recover the plaintext m from ciphertext $c \in \mathbb{Z}_{n^2}^*$:

1) Compute:

$$u = c^\lambda \mod n^2$$

2) Apply the L-function and modular multiplication:

$$m = L(u) \cdot \mu \mod n$$

where:

$$L(u) = \frac{u-1}{n}$$

4) Additive Homomorphism: Paillier encryption satisfies an additive homomorphic property. For two plaintexts $m_1, m_2 \in \mathbb{Z}_n$ and their encryptions $c_1 = \text{Enc}(m_1)$ and $c_2 = \text{Enc}(m_2)$:

$$c_1 \cdot c_2 \mod n^2 = \text{Enc}(m_1 + m_2 \mod n)$$

This allows secure computation of sums on encrypted data, enabling applications in electronic voting, private surveys, and federated data aggregation.

B. Homomorphic Encryption and Computational Implications

Homomorphic encryption (HE) schemes allow operations on ciphertexts that map directly to operations on plaintexts. Paillier belongs to the class of partially homomorphic encryption (PHE) schemes, supporting only addition, but it is significantly more efficient than fully homomorphic encryption (FHE) for many tasks [5].

HE schemes place heavy computational demands on hardware [6] due to their reliance on:

- Large integer modular exponentiation (2048-bit and beyond)
- Multiple modular multiplications per ciphertext operation
- Domain-specific transformations (e.g., Montgomery arithmetic)

Thus, hardware acceleration offers significance performance gains for both encryption and decryption.

C. Hardware Requirements and Motivation

Accelerating Paillier cryptography involves supporting:

- Modular exponentiation over n and n^2
- Repeated Montgomery multiplications
- Domain transformations (e.g., to/from Montgomery domain)
- Efficient operand scheduling and reuse

Each encryption operation involves computing g^m and $r^n \bmod n^2$, while decryption requires $c^\lambda \bmod n^2$ and a modular multiplication with μ. A shared, reusable Montgomery multiplication unit is ideal for servicing both paths. However, achieving high throughput requires pipelining, input alignment, and careful bus arbitration across the datapath.

Our architecture addresses these challenges by implementing a parameterized, reusable, systolic Montgomery multiplication core, integrated into FSM-driven encryption and decryption pipelines.

D. Prior Work in Cryptographic Hardware Acceleration

Significant research has focused on accelerating modular arithmetic for RSA, ECC [11], and FHE schemes [12]. Prior efforts include:

- Systolic Montgomery multipliers [3]
- Reconfigurable cryptoprocessors [15]

While homomorphic schemes like BGV and CKKS have received attention for secure machine learning workloads [13], [14], Paillier has seen limited hardware exploration [16]. Our design targets general-purpose cryptographic acceleration using parameterized, reusable arithmetic blocks amenable to synthesis for large key sizes.

III. Algorithm Discussion

Efficient implementation of modular exponentiation requires a fast and scalable method for computing modular multiplication of large integers. Our accelerator is built around a pipelined Montgomery multiplication unit using the Coarsely Integrated Operand Scanning (CIOS) algorithm. We also implement modular exponentiation using both square-and-multiply and Montgomery ladder strategies to support encryption and decryption paths in the Paillier cryptosystem.

A. Motivation for Montgomery Multiplication

Naive modular multiplication of large integers typically requires explicit division operations to reduce the intermediate result modulo p, which is costly in hardware. Montgomery multiplication [8] avoids division by transforming all operands into a special domain where reduction can be performed using additions and bit-shifts, which are much more hardware-friendly.

Montgomery multiplication is especially attractive for cryptosystems like Paillier and RSA because it allows modular exponentiation to be performed entirely in the Montgomery domain, significantly reducing the overhead of repeated reductions [9].

B. Choice of CIOS Algorithm

Among the various implementations of Montgomery multiplication, we adopt the Coarsely Integrated Operand Scanning (CIOS) method, following the algorithmic structure described in [10], and the processing-element decomposition as presented in [3]. CIOS tightly interleaves the reduction and multiplication steps, allowing partial products to be reduced as they are computed. This reduces memory usage and simplifies control logic compared to separate-stage (SOS) or Finely Integrated Operand Scanning (FIOS) variants.

Benefits of CIOS:

- Lower memory footprint due to integrated reduction
- Simpler control flow, suitable for systolic pipelining
- Well-understood timing and deterministic behavior

Drawbacks:

- Slightly more complex data dependencies across iterations
- Less modular than SOS for reusing multipliers across stages

The CIOS structure is well suited to our systolic architecture, where each iteration maps cleanly to a fixed sequence of processing elements.

C. CIOS Algorithm and PE Mapping

The CIOS algorithm computes $T = a \cdot b \cdot R^{-1} \bmod p$ using the steps shown in Algorithm 1. Each operation within a loop iteration is mapped to a specific hardware unit, allowing pipelined and parallel execution.

Algorithm 1 CIOS algorithm for Montgomery multiplication [10]

Input: $p < 2^K$, $p' = -p^{-1} \bmod 2^w$, w, s, $K = s \cdot w$: bit length, $R = 2^K$, $a, b < p$
Output: $a \cdot b \cdot R^{-1} \bmod p$

$T \leftarrow 0$
for $i \leftarrow 0$ to $s-1$ **do**
  $C \leftarrow 0$
  for $j \leftarrow 0$ to $s-1$ **do**
    $(C, S) \leftarrow T[j] + a[i] \cdot b[j] + C$
    $T[j] \leftarrow S$
  end for
  $(C, S) \leftarrow T[s] + C$
  $T[s+1] \leftarrow C$
  $m \leftarrow T[0] \cdot p' \bmod 2^w$
  $(C, S) \leftarrow T[0] + m \cdot p[0]$
  for $j \leftarrow 1$ to $s-1$ **do**
    $(C, S) \leftarrow T[j] + m \cdot p[j] + C$
    $T[j-1] \leftarrow S$
  end for
  $(C, S) \leftarrow T[s] + C$
  $T[s-1] \leftarrow S$
  $T[s] \leftarrow C$
end for
return T

1) PE Breakdown: Each stage of the inner and outer loop maps to a fixed-function processing element in the systolic datapath [3]:

- α **cell:** Computes partial product $T[j] + a[i] \cdot b[j] + C$
- α_f **cell:** Propagates carry into $T[s+1]$
- β **cell:** Computes reduction constant $m = T[0] \cdot p' \bmod 2^w$
- γ **cell:** Performs correction $T[j] + m \cdot p[j] + C$
- γ_f **cell:** Final carry propagation into $T[s]$, $T[s-1]$

This decomposition enables uniform pipeline timing across each row in the systolic array. A single row of processing elements executes the full set of CIOS operations in a fixed sequence, enabling predictable latency and constant-time operation.

D. Modular Exponentiation Methods

Modular exponentiation is performed by issuing repeated Montgomery multiplications using either a standard square-and-multiply method or a constant-time Montgomery ladder.

1) Square-and-Multiply: This approach scans the exponent from most to least significant bit. For each bit:

- Square the current result
- Multiply by the base if the bit is 1

While efficient, this method exhibits data-dependent memory access and timing, which can be exploited in side-channel attacks.

2) Montgomery Ladder [4]: To mitigate timing-based leakage, we also implement the Montgomery ladder. For each exponent bit, both a square and a multiply are performed, regardless of the bit value. The algorithm maintains two registers, R_0 and R_1, and executes the following steps:

$$\text{If } e_i = 0: \quad R_1 \leftarrow R_0 \cdot R_1, \quad R_0 \leftarrow R_0^2$$

$$\text{If } e_i = 1: \quad R_0 \leftarrow R_0 \cdot R_1, \quad R_1 \leftarrow R_1^2$$

This ensures consistent execution flow and avoids branches or memory-access timing differences. However, our current implementation is not yet hardened against cache-based side-channel attacks; future versions will implement constant-memory-access strategies to ensure full timing uniformity.

IV. Architecture

A. Top Level Design

The accelerator consists of two separate modules, one for encryption and one for decryption, both based on a common Montgomery multiplication unit. The output of each module corresponds to the operations defined previously. Each path consists of a series of modular multiplications, exponentiations, and Montgomery domain transformations, handled by the Montgomery multiplication and modular exponentiation units. The control and dataflow are managed by dedicated FSMs that handle instruction scheduling and operand movement within each module.

Each module is fully parameterized to support a variety of input sizes with minimal modification. The generation of relevant inputs is handled by the software interface. Inputs—large integers of size $S \times W$ bits—are represented as arrays of length S and word width W.

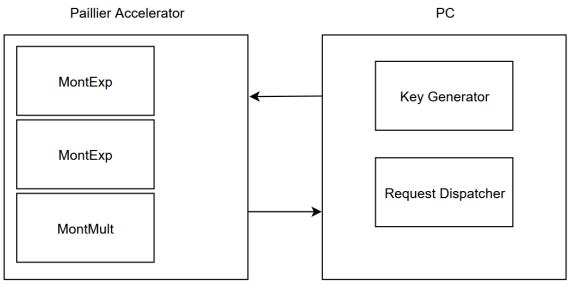

Fig. 1. Top-level architecture of the Paillier accelerator.

TABLE I
INPUTS AND OUTPUTS FOR PAILLIER ENCRYPTION MODULE

Signal	Description
`m[S]`	Plaintext message (m) in Montgomery domain
`g[S]`	Public base g, preconverted to Montgomery domain
`r[S]`	Random $r \in \mathbb{Z}_n^*$ in Montgomery domain
`n[2S]`	Modulus n^2 for encryption
`n_prime`	$-n^{-1} \bmod W$ for Montgomery reduction
`mont_one[2S]`	$R \bmod n^2$ in Montgomery domain
`c[2S]`	Output ciphertext $c = g^m \cdot r^n \bmod n^2$

B. Montgomery Multiplication Unit

The core computational unit is the Montgomery multiplication unit, which is designed as a pipelined systolic array. Our architecture is an adaptation of the design presented in [3], simplifying the control logic by reusing PEs within each row rather than relying on the shifting logic employed in the reference design.

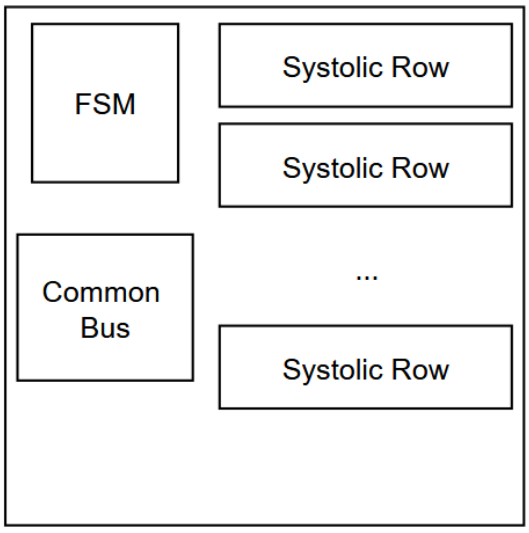

Fig. 2. Montgomery multiplication unit

TABLE II
INPUTS AND OUTPUTS FOR PAILLIER DECRYPTION MODULE

Signal	Description
`c[2S]`	Ciphertext input $c \in \mathbb{Z}_{n^2}^*$
`n[S]`	Modulus n
`n_squared[2S]`	Modulus n^2
`lambda_val[2S]`	Private key $\lambda = \mathrm{lcm}(p-1, q-1)$
`n_prime`	$-n^{-1} \bmod W$
`mont_one[S]`	$R \bmod n$ for Montgomery conversion
`m[S]`	Output plaintext message m

C. Systolic Array Structure

The array consists of a two-dimensional chain of processing elements (PEs), each responsible for a different block in the Montgomery multiplication algorithm. The design uses five distinct PE types—α, β, γ, α_f, and γ_f (explained in detail in the algorithm section)—to segment the operation into well-defined steps that can be cleanly pipelined.

Each row contains one PE of each type, with the number of activations per type varying. For inputs of width W and size S, the α cell is executed S times, and the γ cell $S-1$ times. All other PEs execute once per row. The execution pattern is illustrated in Figure 3. All PEs are designed to execute in three cycles, providing consistent latency throughout the array.

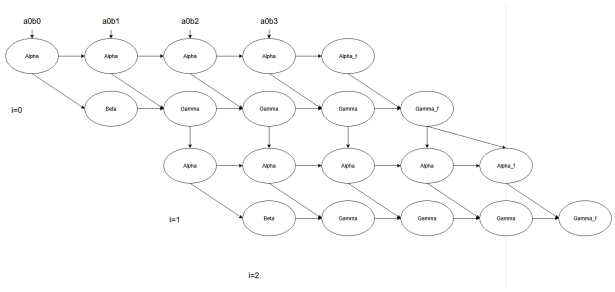

Fig. 3. Systolic Array Architecture for $s = 4$. Reproduced from [3] under the terms of the Creative Commons Attribution 4.0 International License (CC BY 4.0).

Each row in the systolic array corresponds to one iteration of the outer loop, while columns represent parallel execution stages. Rows communicate via a shared bus, exchanging intermediate values produced by inner-loop operations.

D. Bus, Dispatcher, and Timing

Each row is connected to a shared common bus used to read and write intermediate values during Montgomery reduction. A row-local FIFO is used to buffer writes and ensure correct timing and serialization across rows.

A centralized dispatcher coordinates operand injection and stage advancement. It streams operands into the array over handshake-driven buses, issuing one new row after the first γ iteration completes. Active rows are tracked via a global cycle counter. When a row completes execution, its result is flushed into a writeback queue, and the row becomes eligible for reuse.

Cycle-level alignment is enforced through fixed-latency pipelining and valid-ready synchronization. Since the timing

behavior is statically defined, dependent modules—such as the exponentiation FSM—can safely pipeline subsequent operations without introducing feedback hazards or requiring speculative logic.

TABLE III
OPERAND SCHEDULING ACROSS CYCLES (CYCLES 1–10)

	1	2	3	4	5	6	7	8	9	10
α_1	$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$	$A_{0,5}$	$A_{0,6}$	$A_{0,7}$		$A_{3,0}$
α_2				$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	$A_{1,5}$	$A_{1,6}$
α_3							$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
β		β_0			β_1			β_2		β_3
γ_1			$\gamma_{0,1}$	$\gamma_{0,2}$	$\gamma_{0,3}$	$\gamma_{0,4}$	$\gamma_{0,5}$	$\gamma_{0,6}$	$\gamma_{0,7}$	
γ_2						$\gamma_{1,1}$	$\gamma_{1,2}$	$\gamma_{1,3}$	$\gamma_{1,4}$	$\gamma_{1,5}$
γ_3									$\gamma_{2,1}$	$\gamma_{2,2}$
α_f									$\alpha_f 0$	
γ_f										$\gamma_f 0$

E. Parameterization and Scalability

The architecture is designed to be fully scalable across different operand sizes by adjusting three parameters: the number of rows (N), the number of cells per row (S), and the word width (W). These parameters are configured at generation time, allowing the same RTL backbone to support 1024-, 2048-, or 4096-bit keys without structural changes. Parameter propagation is handled systematically across control and datapath units to ensure timing consistency.

F. Modular Exponentiation Unit

Modular exponentiation is implemented using the Montgomery ladder algorithm described in Section **??**. This unit is central to both encryption and decryption flows and is built to reuse the shared Montgomery multiplier.

A centralized FSM governs the exponentiation flow. It receives exponent bits in most-significant-bit-first order, initializes operand registers, and issues Montgomery multiplication commands to the dispatcher. Each multiplication uses a start/done handshake protocol, enabling deterministic sequencing without stalling the wider datapath.

Internally, the FSM maintains state registers and multiplexers for operand selection. Both square and multiply operations are scheduled every cycle regardless of the exponent bit, ensuring uniform timing and constant-time execution. Although all multiplications are serialized through a shared multiplier, the exponentiation controller is pipelined at the control level.

All inputs and outputs remain in Montgomery form. Domain transitions (i.e., conversion into and out of Montgomery space) are handled externally. Intermediate ladder state remains in Montgomery space throughout, reducing conversion overhead.

The unit supports arbitrary exponent lengths and is fully parameterized over input size (number of limbs S and word width W). This allows efficient handling of short (e.g., 256-bit) and long (e.g., 2048-bit) exponents using the same logic.

G. Unit Reuse

The same modular exponentiation unit is reused across both the encryption and decryption modules. Operand initialization and configuration are handled externally, and the exponentiation FSM itself is agnostic to operand semantics. This reusability minimizes code duplication and allows tightly integrated operand and control flow between encryption (g^m, r^n) and decryption (c^λ) operations.

The Montgomery multiplier is also reused across all modular arithmetic operations. Operand scheduling and resource sharing are handled through the dispatcher, which allows serial reuse while maintaining correct execution order and timing.

V. EVALUATION AND SIMULATION

A. Methodology Overview

We evaluate our accelerator through cycle-accurate simulation, multi-level functional verification, and detailed microarchitectural timing analysis. Although post-synthesis evaluation (e.g., area, power, and physical timing closure) is left for future work, the current RTL demonstrates deterministic behavior, modular correctness, and verifiable end-to-end cryptographic operation.

We used Verilator to compile and simulate SystemVerilog RTL into C++ test harnesses and ModelSim for rapid verification of individual models. Test inputs and result validation were performed by a Python verification script.

B. Verification Methodology

We performed functional verification across three abstraction levels:

1) **Unit-level testing**: All individual Montgomery pipeline elements (α, β, γ, α_f, γ_f) were tested independently for correct arithmetic and latency behavior.
2) **Submodule testing**: The complete Montgomery multiplier and modular exponentiator were verified using directed and randomized operand pairs. Each result was compared against a reference GMP-based software implementation.
3) **Full-system testing**: End-to-end correctness of the full encryption and decryption flow was verified. We confirmed that messages passed through the $m \to c \to m'$ roundtrip matched the original plaintext for randomized key pairs.

C. Simulation Infrastructure

Key aspects of our simulation framework include:

- **Cycle-accurate simulation:** Verilator traces enabled fine-grained inspection of control signals, operand buses, and valid-ready timing through the pipeline.
- **Waveform inspection:** Value Change Dump (VCD) waveforms were generated for all tests, enabling FSM tracking and debugging of multicycle stalls.
- **Self-checking testbenches:** Assertions in both Verilog and C++ monitored FSM state transitions, operand validity, and stall behavior.

D. Cycle-Level Results

In our baseline RTL implementation, a single Montgomery multiplication with $S = 32$, $W = 64$ completes in approximately **640 cycles**. This reflects conservative dispatching, FIFO contention, and full valid-ready handshaking across the pipeline.

While suboptimal compared to fully optimized datapaths, our implementation emphasizes testability, reuse, and functional determinism. Moreover, this architecture shares core design patterns with high-throughput public-key accelerators.

E. Optimization Plan and Target Performance

We estimate that the optimized RTL could perform a 2048-bit Montgomery multiplication in approximately 150–160 cycles [3]. This improvement is based on the following targeted refinements:

- **Tighter array timing:** By decoupling read/write operations from the shared bus and enabling independent propagation, we eliminate stalls between pipeline stages.
- **Aligned row activation:** Fine-tuned cycle offsets between systolic rows allow new operations to enter the array at every cycle, maximizing throughput.
- **Improved bus scheduling:** Parallelizing output handling and redesigning the writeback queue allows continuous read-back without inter-stage blocking.

All proposed optimizations maintain full compatibility with the current datapath and require only modifications to control logic and bus interface timing. No changes are needed to the core PE architecture, enabling rapid iteration and retiming.

F. Summary

Our simulation campaign confirms:

- Deterministic execution and full-cycle predictability for each operation.
- Bit-exact agreement with reference models across randomized input spaces.
- Clear opportunities for optimization that preserve design modularity.

We conclude that the current design is a stable foundation for subsequent physical realization and layout synthesis, while maintaining a clean control/data separation and high portability across operand sizes.

VI. Discussion

A. Deployment Use Cases

The proposed accelerator targets secure, latency-sensitive workloads that benefit from encrypted-domain computation. Applications include federated learning [17], encrypted machine learning [18], and encrypted databases [19].

Parameterization across key sizes (1024, 2048, 4096 bits) supports a range of security levels while maintaining a consistent hardware backend. Hardware execution of both encryption and decryption eliminates high-overhead software loops, enabling fast, secure computation.

B. Architectural Limitations and Bottlenecks

Despite significant improvements over software baselines, the current architecture has several performance-limiting factors. Chief among these is the shared Montgomery multiplication unit, which handles all modular arithmetic operations sequentially. This serialization simplifies control logic and conserves area but reduces concurrency when processing simultaneous encryption and decryption tasks.

Additionally, the centralized memory bus can become a performance bottleneck due to contention during parallel operand read/write operations. The bit-serial nature of the exponentiation controller, while efficient for deterministic timing, restricts throughput for long exponent computations by enforcing a strictly sequential execution model.

C. Design Tradeoffs and Architectural Rationale

The current design reflects deliberate tradeoffs prioritizing modularity, determinism, and security over raw throughput. Unified arithmetic hardware for encryption and decryption reduces area and verification complexity but limits parallel execution. Parameterization improves flexibility across operand sizes but introduces FSM and scheduling complexity.

Deterministic timing behavior, while imposing control rigidity, improves side-channel resistance by minimizing observable execution variation [4]. Overall, the design favors analyzability, reusability, and secure integration over specialization.

VII. Conclusion and Future Directions

A. Summary of Contributions

This work presents a cycle-deterministic hardware accelerator for Paillier encryption and decryption. It enables additive homomorphic operations via a shared datapath architecture built around a systolic Montgomery multiplier and a modular exponentiation controller. The system achieves a speedup over software based execution, supports parameterized operand sizes, and provides a secure, scalable platform for privacy-preserving computation.

B. Opportunities for Optimization

Multiple avenues exist for future refinement. These include parallelizing the exponentiation process via windowed techniques or multi-instance controllers, restructuring the memory bus for reduced contention, and implementing pipelined dispatchers to improve operand throughput. FPGA/ASIC synthesis and physical benchmarking would enable assessment of power, area, and thermal profiles. Hardware/software co-design could further enhance flexibility through dynamic key management and offloaded control routines.

C. Security Enhancement Roadmap

While the current design enforces cycle-level determinism, future iterations could strengthen side-channel resistance by incorporating constant-memory-access schemes, power-equalizing datapaths, and on-chip entropy sources for masking. These additions would enhance robustness in adversarial settings and elevate the design's suitability for secure edge deployment [20].

VIII. Role of AI in Accelerator Design and Development

Artificial intelligence (AI), particularly in the form of large language models (LLMs), played an instrumental role in various stages of this project's development. From early design prototyping to code generation, verification planning, and documentation, AI-assisted workflows enabled gains in productivity, scalability, and ideation.

A. Design Assistance and Automation

Throughout the RTL development cycle, AI tools were used to scaffold components, auto-generate repetitive Verilog modules, and assist with initial integration logic. These generative tools reduced implementation time and improved turnaround during architecture iteration. AI was also leveraged to cross-check implementation patterns against known best practices, such as pipelining for fixed-latency execution or systolic array layout strategies.

B. Debugging and Verification Support

AI-based assistants were employed to reason about failing testbench outputs, identify logic inconsistencies, and propose assertions for formal property checks. AI also helped perform code reviews and identify errors in logic or syntax and to map out FSM logic. Additionally, Python and C++ based verification scripts for operand generation and output decoding were partially AI-generated, accelerating the development of a robust functional testing framework.

C. Benefits

AI significantly lowered the barrier for rapid iteration and parallel development, allowing hardware design, documentation, and testing to progress concurrently. It supported fast prototyping and reduced the overhead of tedious design tasks, enabling more focus on architectural decisions and pipeline analysis. Moreover, the AI excelled at providing quick overviews and summaries, easily synthesizing information, especially in niche areas like systolic cryptographic arithmetic.

D. Limitations and Risks

Despite these benefits, the use of AI in hardware design came with caveats. Generative models often produced faulty logic and commonly made repetitive errors. AI-generated testbenches and FSMs often required manual correction to satisfy strict timing and synthesis constraints. Furthermore, LLMs occasionally suggested outdated or incompatible constructs. Lastly, while AI accelerated drafting and formatting, it occasionally introduced stylistic inconsistency or verbosity that had to be revised for publication standards.

E. Outlook

Overall, AI was a valuable augmentative tool, not a replacement for hardware engineering expertise. Its integration into the development cycle improved iteration speed, enhanced verification coverage, and streamlined the communication of complex design decisions. However, using these tools required significant human oversight and verification, highlighting the need for humans and AI to work in tandem.

References

[1] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," Lecture Notes in Computer Science, vol. 1592, pp. 223–238, Jan. 1999. doi:10.1007/3-540-48910-x_16

[2] S. J. Mohammed and D. B. Taha, "Performance Evaluation of RSA, ElGamal, and Paillier Partial Homomorphic Encryption Algorithms," 2022 International Conference on Computer Science and Software Engineering (CSASE), Duhok, Iraq, 2022, pp. 89-94, doi: 10.1109/CSASE51777.2022.9759825.

[3] A. Mrabet et al., "High-performance elliptic curve cryptography by using the CIOS method for modular multiplication," Lecture Notes in Computer Science, pp. 185–198, 2017. doi:10.1007/978-3-319-54876-0_15

[4] M. Joye and S.-M. Yen, "The Montgomery Powering Ladder," Lecture Notes in Computer Science, pp. 291–302, 2003. doi:10.1007/3-540-36400-5_22

[5] V. Biksham and D. Vasumathi, "Query based computations on encrypted data through homomorphic encryption in Cloud computing security," 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), pp. 3820–3825, Mar. 2016. doi:10.1109/iceeot.2016.7755429

[6] Y. Gong et al., "Practical solutions in fully homomorphic encryption: A survey analyzing existing acceleration methods," Cybersecurity, vol. 7, no. 1, Mar. 2024. doi:10.1186/s42400-023-00187-4

[7] S. Gueron, "Efficient software implementations of modular exponentiation," Journal of Cryptographic Engineering, vol. 2, no. 1, pp. 31–43, Apr. 2012. doi:10.1007/s13389-012-0031-5

[8] P. L. Montgomery, "Modular multiplication without trial division," Mathematics of Computation, vol. 44, no. 170, p. 519, Apr. 1985. doi:10.2307/2007970

[9] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, Handbook of Applied Cryptography. Boca Raton: CRC Press, 2018.

[10] C. Kaya Koc, T. Acar, and B. S. Kaliski, "Analyzing and comparing montgomery multiplication algorithms," IEEE Micro, vol. 16, no. 3, pp. 26–33, Jun. 1996. doi:10.1109/40.502403

[11] *ECC/RSA Public Key Accelerators*, Synopsys Inc., Mountain View, CA, USA. [Online]. Available: https://www.synopsys.com/designware-ip/security-ip/cryptography-ip/public-key-accelerators/ecc-rsa-public-key-accelerators.html

[12] M. Zhou et al., "FHEMEM: A processing in-memory accelerator for fully homomorphic encryption," IEEE Transactions on Emerging Topics in Computing, pp. 1–16, 2023. doi:10.1109/tetc.2025.3528862

[13] R. Geelen et al., "BASALISC: Programmable hardware accelerator for BGV fully homomorphic encryption," IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 32–57, Aug. 2023. doi:10.46586/tches.v2023.i4.32-57

[14] S. Fan, X. Deng, Z. Tian, Z. Hu, L. Chang, R. Hou, D. Meng, and M. Zhang, "Taiyi: A high-performance CKKS accelerator for practical fully homomorphic encryption," arXiv preprint arXiv:2403.10188, 2024. [Online]. Available: https://arxiv.org/abs/2403.10188

[15] L. Liu, B. Wang, and S. Wei, Reconfigurable Cryptographic Processor. Singapore: Springer, 2018.

[16] Z. Yang, "FPGA-based hardware acceleration of homomorphic encryption for federated learning," M.Phil. thesis, Dept. of Computer Science and Engineering, Hong Kong Univ. of Sci. and Technol., Hong Kong, Aug. 2020.

[17] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, "BatchCrypt: Efficient homomorphic encryption for cross-silo federated learning," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2020, pp. 493–506.

[18] K. Muhammad, K. A. Sugeng, and H. Murfi, "Machine learning with partially homomorphic encrypted data," Journal of Physics: Conference Series, vol. 1108, p. 012112, Nov. 2018. doi:10.1088/1742-6596/1108/1/012112

[19] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, CryptDB: Protecting Confidentiality with Encrypted Query Processing. Proceedings of the 23rd ACM Symposium on Operating Systems Principles, pp. 85 - 100, 2011.

[20] M. Kaleem, M. Mushtaq, A. Ramay, A. Mahmood, T. Khan, S. Hussain, A. Anwar, H. Bhatti, and M. Azhar, "Navigating side-channel attacks: A comprehensive overview of cryptographic system vulnerabilities," *J. Comput. Biomed. Inform.*, vol. 7, 2024.