Hardware Accelerator for Winograd-based NTT using Generalized KRD Montgomery Algorithm

Meher Narula (230648)
Department of Electrical Engineering
Indian Institute of Technology, Kanpur

Supervisors:
Prof. Debapriya Basu Roy & Prof. Chithra

Collaborator:
Saksham Verma (230899)

Academic Year 2025-2026

Abstract

This project addresses key computational challenges in Homomorphic Encryption (HE) systems, where polynomial multiplication via the Number Theoretic Transform (NTT) remains a primary performance bottleneck. Building upon the Winograd NTT approach—which reduces the number of modular multipliers required for higher–radix implementations—we develop a generalized *KRD Montgomery reduction* algorithm that overcomes the parameter–specific limitations of existing methods.

Our work extends the KRED reduction technique for sparse primes of the form

$$2^n \pm 2^m \pm 1,$$

commonly used in FALCON and DILITHIUM, and transitions to a more systematic KRD Montgomery framework for broader applicability.

The proposed architecture supports arbitrary cryptographic primes with complex Canonical Signed Digit (CSD) representations through automated Verilog code generation.

Extensive validation using XILINX VIVADO confirms perfect functional correctness across a wide range of parameter sets. Furthermore, the developed solution demonstrates consistent performance advantages—both in resource utilization and timing characteristics—when compared to specialized, parameter–restricted implementations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Context

The increasing demand for secure cloud computing and privacy-preserving machine learning has brought Homomorphic Encryption (HE) to the forefront of cryptographic research. HE enables computation on encrypted data without decryption, maintaining data confidentiality throughout processing. However, this capability comes at the cost of significant computational overhead, particularly in polynomial multiplication operations that form the mathematical foundation of most HE schemes.

The Number Theoretic Transform (NTT) serves as the primary acceleration technique for polynomial multiplication in lattice-based cryptography, reducing the complexity from $O(n^2)$ to $O(n \log n)$. Despite this improvement, NTT remains computationally intensive, especially for the large polynomial degrees required by modern security standards. This computational burden has motivated extensive research into hardware acceleration of NTT operations.

1.2 Comprehensive Background Analysis

1.2.1 Winograd NTT: Mathematical Foundation and Advantages

The Winograd NTT approach, introduced by Mandal and Roy, represents a significant departure from conventional Cooley-Tukey and Gentleman-Sande architectures. This method builds on Winograd's pioneering work in signal processing algorithms, adapting his mathematical insights to the domain of cryptographic polynomial multiplication.

Key characteristics of the Winograd NTT approach include:

- **Reduced Multiplier Count:** For higher radix implementations (radix-8, radix-16, etc.), Winograd NTT requires significantly fewer modular multipliers compared to traditional approaches. This reduction comes from exploiting mathematical symmetries and common subexpression elimination in the transformation matrices.

- **Mathematical Reformulation:** Winograd NTT re-expresses the NTT computation using Chinese Remainder Theorem principles, decomposing the transformation into smaller, more efficient computational units. This reformulation enables better resource sharing and reduces the overall arithmetic complexity.

- **Hardware Optimization:** The approach specifically targets hardware implementations, with optimized architectures for different cryptographic standards:
  - **CRYSTALS-Dilithium:** Radix-16 implementation for 256-degree polynomials
  - **FALCON:** Radix-8 implementation for 512-degree polynomials
  - **CRYSTALS-Kyber:** Configurable radix-16/parallel radix-8 for mixed-radix computation
- **K-RED Integration:** The Winograd approach incorporates specialized K-RED modular multiplication techniques optimized for specific modulus structures, providing additional performance gains for target cryptographic schemes.

Despite these advantages, the Winograd approach shares the same fundamental limitation as other specialized implementations: it requires manual optimization and algorithm derivation for each new parameter set, making it unsuitable for cryptographic agility.

1.2.2 Proteus Architecture: Design-Time Flexibility

The Proteus architecture, developed by Hirner et al., addresses the parameter flexibility challenge through a design-time configurable approach. Rather than supporting runtime parameter changes, Proteus generates optimized hardware architectures tailored to specific polynomial degrees and coefficient modulus sizes provided during the design phase.

Key features of the Proteus architecture include:

- **Dual Architecture Support:** Proteus can generate both Single-Path Delay Feedback (SDF) and Multi-path Delay Commutator (MDC) NTT architectures, allowing designers to choose between area efficiency and performance based on application requirements.
- **Word-Level Montgomery Reduction:** The architecture employs an optimized word-level Montgomery reduction algorithm that efficiently maps to FPGA DSP units. This implementation breaks down the reduction into word-sized operations that align with native DSP capabilities.
- **Parametric Hardware Generation:** For given parameters (polynomial size n, coefficient modulus size $\log_2 q$), Proteus automatically generates the complete NTT hardware including:
  - Butterfly Unit (BFU) configurations
  - Memory addressing schemes
  - Twiddle factor storage and access patterns
  - Control logic for different NTT operations
- **Performance Claims:** The authors report 1.8× speedup over existing implementations while using 1.75× fewer DSPs and 3× fewer BRAMs compared to state-of-the-art SDF-based NTT implementations.

However, Proteus still requires careful DSP mapping and offline pre-computation for each parameter set, and its word-level Montgomery approach introduces complexity that limits true parameter generality.

1.3 Problem Statement and Research Gap

The comprehensive analysis of existing approaches reveals a fundamental tension in NTT hardware design: the trade-off between specialization and generality. Specialized implementations like Winograd NTT achieve excellent performance for specific parameters but lack adaptability. Flexible approaches like Proteus support multiple parameters but require complex hardware mapping and lose the optimization benefits of specialization.

This project addresses the critical research question: *Can we develop an NTT arithmetic unit that maintains the performance advantages of specialized implementations while achieving true parameter generality?*

The specific challenges include:

- Developing mathematical generalizations of reduction algorithms beyond specific parameter sets
- Creating automated tools that eliminate manual algorithm derivation
- Supporting arbitrary primes with complex structural properties
- Maintaining performance consistency across diverse parameter spaces
- Ensuring practical implementability with existing EDA tools and workflows

1.4 Our Contributions

This work makes several key contributions to the field of cryptographic hardware acceleration:

- **Algorithm Generalization:** We successfully generalized the KRed algorithm for sparse primes of the form $2^n \pm 2^m \pm 1$, demonstrating that mathematical patterns in cryptographic primes can be systematically exploited beyond specific cases.
- **Systematic KRD Montgomery:** We developed a parameterized KRD Montgomery algorithm that works for arbitrary primes, using a pure shift-based implementation with CSD decomposition to eliminate sequential operators.
- **Automation Framework:** We created a Python-based automation framework that generates optimized Verilog implementations for any given parameters q and n, significantly reducing development time from days to minutes per parameter set.
- **Comprehensive Validation:** We conducted extensive testing across standardized cryptographic primes and randomized test cases, demonstrating consistent performance and functional correctness.

1.5 Report Organization

This report is organized as follows: Chapter 2 provides detailed methodology covering both algorithm development and implementation aspects. Chapter 3 presents experimental results and comparative analysis. Chapter 4 discusses limitations and future work directions, while Chapter 5 concludes with key findings and implications.

Chapter 2

Methodology

2.1 System Architecture Overview

The overall NTT accelerator follows a pipelined architecture with multiple Butterfly Units (BFUs) operating in parallel. Each BFU contains the core modular multiplication block that forms the computational bottleneck. Our work specifically targets this critical component, developing optimized implementations that can be efficiently integrated into the larger NTT framework.

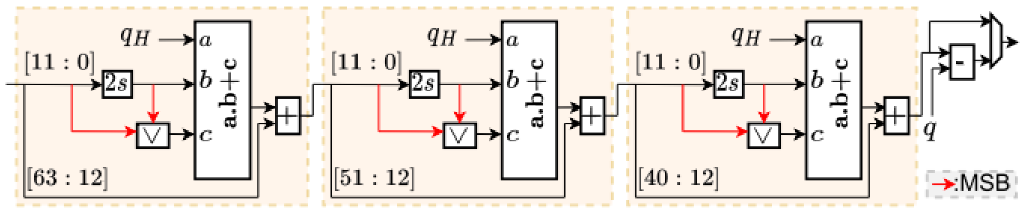


Fig. 9. WL Montgomery reduction circuit for $\log_2 q = 32$ and $w = 12$.

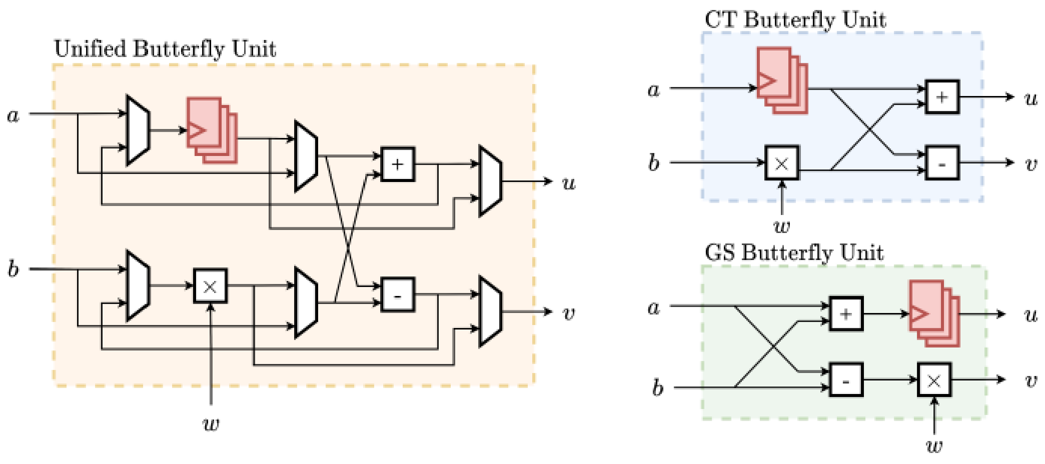


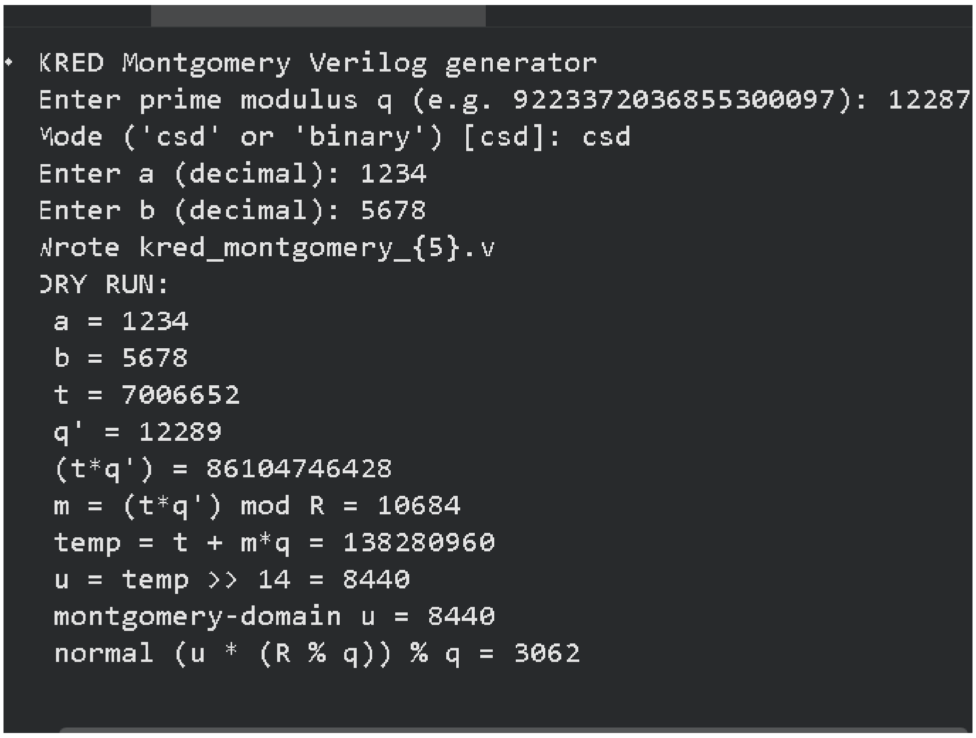
Figure 2.1: High-level architecture of the NTT accelerator showing modular multiplier integration

The architectural decisions were guided by several key considerations:

- **Modularity:** The multiplier design must interface cleanly with existing BFU architectures
- **Parameterization:** The implementation should support runtime or compile-time parameter changes
- **Performance Consistency:** The design should maintain predictable timing across different parameter sets
- **Tool Compatibility:** The implementation must work reliably with standard EDA tools and workflows

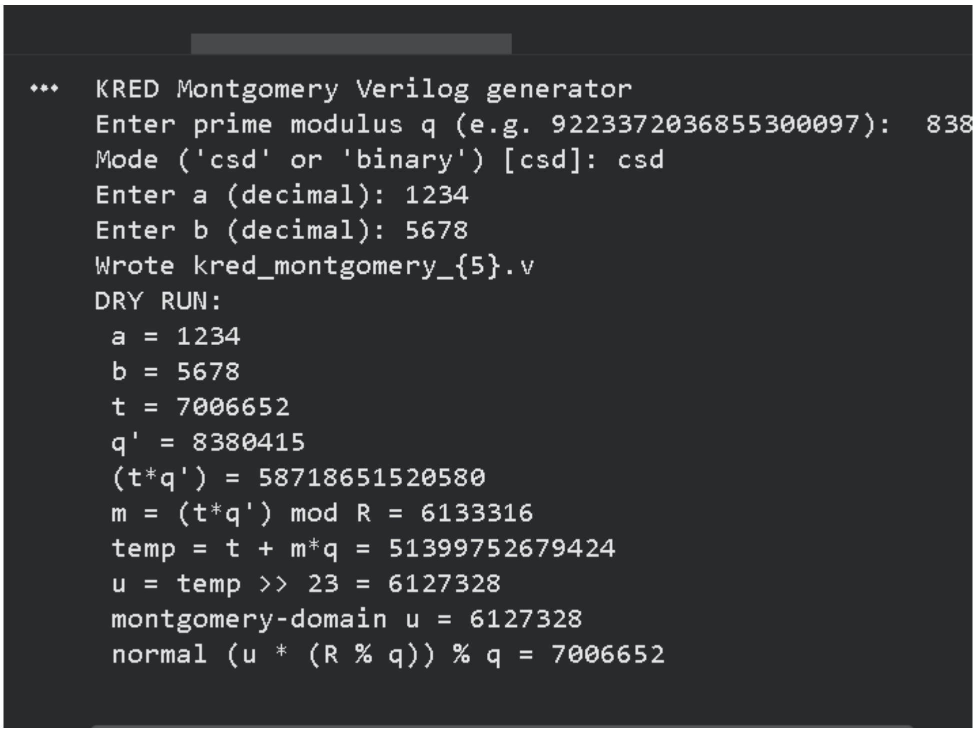
2.2 Automation Framework Implementation

Our Python-based automation framework bridges mathematical algorithms and hardware implementation, generating optimized Verilog code for arbitrary cryptographic parameters.



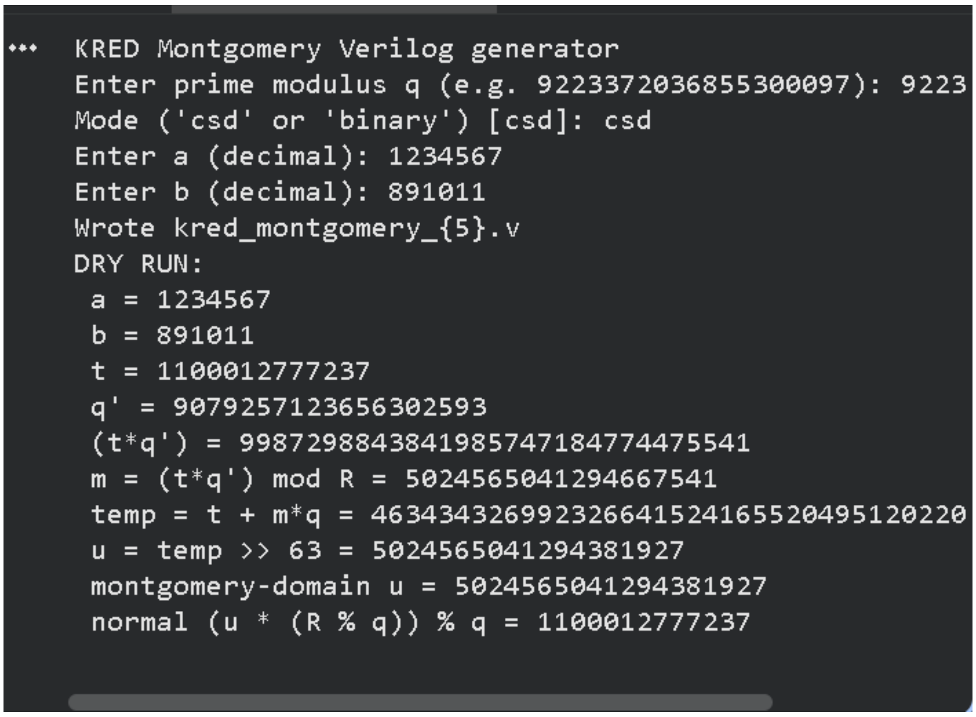
```
• KRED Montgomery Verilog generator
  Enter prime modulus q (e.g. 9223372036855300097): 12287
  Mode ('csd' or 'binary') [csd]: csd
  Enter a (decimal): 1234
  Enter b (decimal): 5678
  Wrote kred_montgomery_{5}.v
  DRY RUN:
   a = 1234
   b = 5678
   t = 7006652
   q' = 12289
   (t*q') = 86104746428
   m = (t*q') mod R = 10684
   temp = t + m*q = 138280960
   u = temp >> 14 = 8440
   montgomery-domain u = 8440
   normal (u * (R % q)) % q = 3062
```

Figure 2.2: Python automation framework output for FALCON parameters (q = 12289)



```
•••  KRED Montgomery Verilog generator
     Enter prime modulus q (e.g. 9223372036855300097):  838
     Mode ('csd' or 'binary') [csd]: csd
     Enter a (decimal): 1234
     Enter b (decimal): 5678
     Wrote kred_montgomery_{5}.v
     DRY RUN:
      a = 1234
      b = 5678
      t = 7006652
      q' = 8380415
      (t*q') = 58718651520580
      m = (t*q') mod R = 6133316
      temp = t + m*q = 51399752679424
      u = temp >> 23 = 6127328
      montgomery-domain u = 6127328
      normal (u * (R % q)) % q = 7006652
```

Figure 2.3: Python automation framework output for Dilithium parameters (q = 8380417)



```
•••  KRED Montgomery Verilog generator
     Enter prime modulus q (e.g. 9223372036855300097): 9223
     Mode ('csd' or 'binary') [csd]: csd
     Enter a (decimal): 1234567
     Enter b (decimal): 891011
     Wrote kred_montgomery_{5}.v
     DRY RUN:
      a = 1234567
      b = 891011
      t = 1100012777237
      q' = 9079257123656302593
      (t*q') = 99872988438419857471847744755541
      m = (t*q') mod R = 5024565041294667541
      temp = t + m*q = 46343432699232664152416552049512022
      u = temp >> 63 = 5024565041294381927
      montgomery-domain u = 5024565041294381927
      normal (u * (R % q)) % q = 1100012777237
```

Figure 2.4: Python automation framework handling large prime modulus (q = 9223372036855300097)

2.3 KRed Algorithm Analysis and Generalization

2.3.1 FALCON KRed Algorithm (Winograd Implementation)

The Winograd paper presents a highly specialized KRed implementation for FALCON parameters ($q = 12289$), with algorithm steps tightly coupled to the specific modulus structure:

Algorithm 1 Modified KRed Algorithm for FALCON (from Winograd Paper)

```
Require: $C = a[13:0] \times b[15:0]$
Ensure: $K'_{FALCON} \times C \mod q$, where $K'_{FALCON} = 512^{-1}$
1: $c_8 = C \ll 3$
2: $cl_1 = c_8[13:0],\ ch_1 = c_8 \gg 14,\ ch_{11} = (ch_1 \ll 12) - ch_1$
3: $cl_2 = ch_{11}[13:0],\ ch_2 = ch_{11} \gg 14,\ ch_{21} = (ch_2 \ll 12) - ch_2$
4: $cl_3 = ch_{21}[13:0],\ ch_3 = ch_{21} \gg 14,\ ch_{31} = (ch_3 \ll 12) - ch_3$
5: $cl_4 = ch_{31}[13:0],\ ch_4 = ch_{31} \gg 14,\ ch_{41} = (ch_4 \ll 12) - ch_4$
6: $cl_5 = ch_{41}[13:0],\ ch_5 = ch_{41} \gg 14,\ ch_{51} = (ch_5 \ll 12) - ch_5$
7: $c_1 = ch_{51} + c1_1 + cl_2 + cl_3 + cl_4 + cl_5$
8: return K-RED($c_1$)
```

This implementation demonstrates several characteristic features of specialized KRed algorithms:

- Fixed shift amounts (3, 14, 12) derived from the specific modulus properties
- Multiple decomposition stages tailored to the bit pattern of $q = 12289$
- Hardcoded bit ranges for intermediate values
- Manual optimization of the reduction chain

2.3.2 KRed Montgomery for FALCON

The Winograd paper also presents a Montgomery variant specifically for FALCON:

Algorithm 2 KRed Montgomery for FALCON (from Winograd Paper)

```
Require: $t = a[13:0] \times b[15:0]$, where $a, b \notin$ Montgomery Domain
Ensure: $t \cdot R^{-1} \mod q$
1: $t_d = (t \ll 14) - (t \ll 12) - t,\ m = t_d[17:0]$
2: $mult_n = (m \ll 14) - (m \ll 12) + m$
3: $res_1 = (t + mult_n) \gg 18$
4: if $result > prime$ then
5:     $result = result - prime$
6: end if
7: return $result$
```

2.3.3 Our KRed Generalization for Sparse Primes

Building on these specialized implementations, we developed a generalized approach for primes of the form $2^n \pm 2^m \pm 1$:

```python
def determine_dilithium_shifts_from_q(q_value: int) -> tuple[int, int]:
    """
    Derives reduction parameters for primes of form 2^N - 2^M + 1
    This generalization enables automated KRed implementation for
    entire families of cryptographic primes with sparse forms
    """
    N_EXPONENT = math.ceil(math.log2(q_value))
    two_n = 2**N_EXPONENT
    two_m_minus_one = two_n - q_value
    two_m = two_m_minus_one + 1

    # Validate the prime structure
    if (two_m & (two_m - 1)) != 0:
        raise ValueError(f"Modulus {q_value} doesn't match sparse form")

    M_EXPONENT = int(math.log2(two_m))
    return N_EXPONENT, M_EXPONENT
```

Listing 2.1: KRed Parameter Derivation for Sparse Primes

This generalization represents a significant step beyond parameter-specific implementations, enabling automated handling of entire prime families used in lattice-based cryptography.

2.4 Transition to KRD Montgomery Approach

Despite successful KRed generalization, several fundamental limitations motivated our transition to the KRD Montgomery approach:

2.4.1 Limitations of KRed Approach

- **Algorithmic Complexity:** Each prime structure requires custom derivation of reduction steps and shift amounts
- **Mathematical Burden:** Implementation demands deep understanding of number theory and modular arithmetic
- **Control Overhead:** Complex state machines needed for different reduction patterns
- **Limited Scope:** Difficult to extend beyond primes with specific sparse forms
- **Development Time:** Significant manual effort required for each new parameter set

2.4.2 Advantages of KRD Montgomery

The KRD Montgomery approach offers several systematic advantages:

- **Uniform Procedure:** Same algorithm steps apply to all primes, with only parameters changing
- **Mathematical Simplicity:** Based on well-established Montgomery multiplication principles
- **Automation Friendly:** Parameters can be computed automatically for any modulus
- **Proven Correctness:** Mathematical foundation ensures functional correctness
- **Performance Predictability:** Consistent behavior across different parameter sets

2.5 Generalized KRD Montgomery Implementation

2.5.1 Mathematical Foundation

Our generalized KRD Montgomery multiplication builds on the standard Montgomery formulation:

$$\text{Montgomery}(a,b) = a \times b \times R^{-1} \mod q \tag{2.1}$$

where $R = 2^k$ with $k = \lceil \log_2(q) \rceil$, and $q' = -q^{-1} \mod R$.

The key insight in our implementation is the use of pure combinational logic based on shift operations and CSD decomposition, eliminating sequential operators entirely.

Algorithm 3 Generalized KRD Montgomery for Arbitrary Primes

```
Require: $a, b, q$ where $a, b < q$
Ensure: $a \times b \times R^{-1} \mod q$
 1: $k \leftarrow \lceil \log_2(q) \rceil$ {Determine word size from modulus}
 2: $R \leftarrow 2^k$ {Montgomery radix}
 3: $q_{prime} \leftarrow -q^{-1} \mod R$ {Precomputed constant}
 4: $t \leftarrow a \times b$ {Full precision multiplication}
 5: $m \leftarrow (t \times q_{prime}) \mod R$ {CSD-based computation}
 6: $u \leftarrow (t + m \times q) \gg k$ {Shift-based reduction}
 7: if $u \geq q$ then
 8:     $u \leftarrow u - q$ {Final reduction if needed}
 9: end if
10: return $u$ {Result in Montgomery domain}
```

2.5.2 CSD Decomposition for Hardware Efficiency

The critical innovation in our implementation is the use of Canonical Signed Digit (CSD) decomposition to enable pure shift-based arithmetic:

```python
def naf_decompose(n: int):
    """
    Non-adjacent form (NAF) decomposition for CSD representation
    Enables multiplication using only shifts and additions/subtractions
    Each term represents (shift_amount,  1 ) for hardware implementation
    """
    terms = []
    k = n
    i = 0
    while k:
        if k & 1:
            # Choose digit to minimize non-zero terms
            if (k & 3) == 1:
                zi = 1
            else:
                zi = -1
            terms.append((i, zi))
            k = k - zi
        k >>= 1
        i += 1
    return terms  # List of (shift,  1 ) for shift-based implementation
```

Listing 2.2: CSD Decomposition for Shift-Based Implementation

This decomposition transforms multiplications into sequences of shifts and additions, enabling highly efficient hardware implementation without sequential operators.

2.5.3 Automated Verilog Generation

Our Python automation framework generates optimized Verilog code with conservative bit-widths to prevent overflow:

```
# Conservative bit-width computation for shift-based implementation
q_bits = q.bit_length()
WIDTH = q_bits
T_BITS = q_bits * 2 + 4         # Product a*b for shift operations
T_D_BITS = T_BITS + WIDTH + 16 # Extended precision for CSD shifts
MULT_BITS = T_BITS + q_bits + 16 # Multiplication accumulator
SUM_BITS = max(T_BITS, MULT_BITS) + 16 # Sum accumulator
U_BITS = SUM_BITS - WIDTH + 16 # Final result width
```

Listing 2.3: Bit-width Computation for Safe Arithmetic

The generated Verilog implements a pure combinational design using shift-and-add operations based on CSD decomposition, providing excellent timing characteristics and resource utilization.

2.6 Verification Methodology

2.6.1 Multi-Level Validation Approach

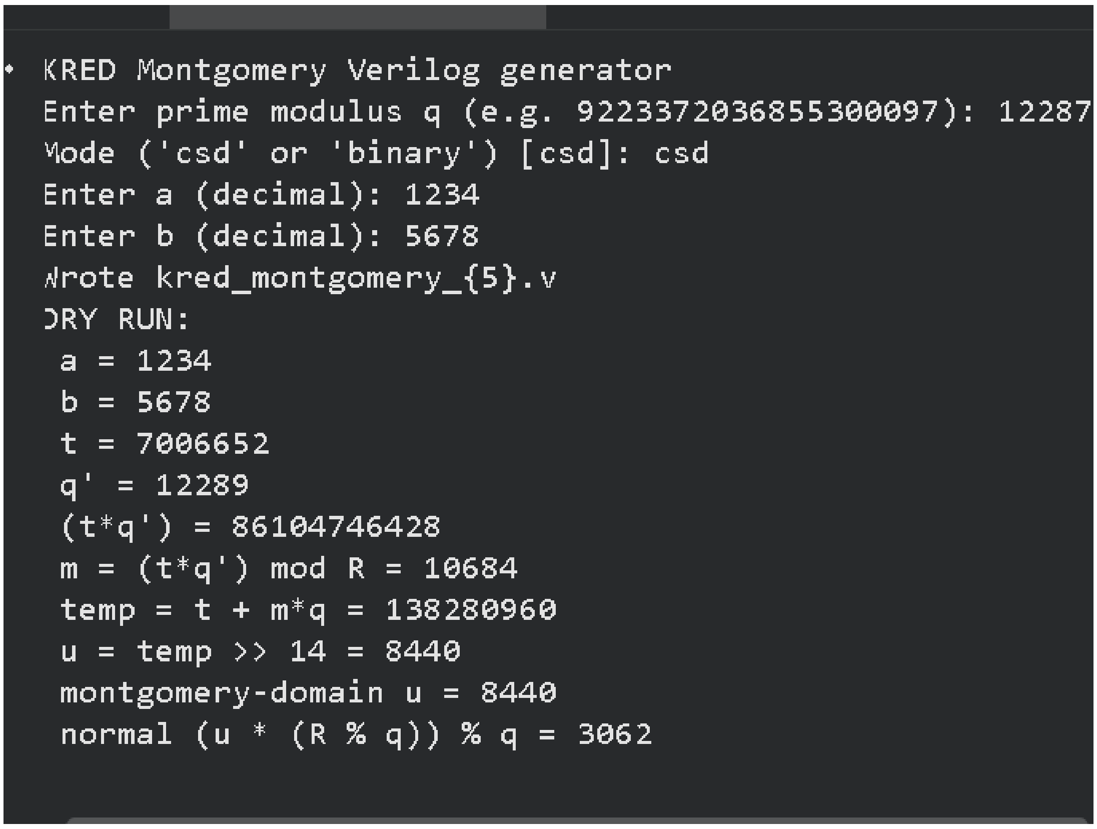
We implemented a comprehensive verification strategy to ensure functional correctness:

- **Mathematical Validation:** Python/Sage implementations for reference calculations
- **Domain Conversion:** All Montgomery results converted to normal domain using $result \times (R \mod q) \mod q$ before comparison

- **Edge Case Testing:** Boundary values, maximum inputs, and special patterns
- **Parameter Coverage:** Standardized primes and randomized test vectors

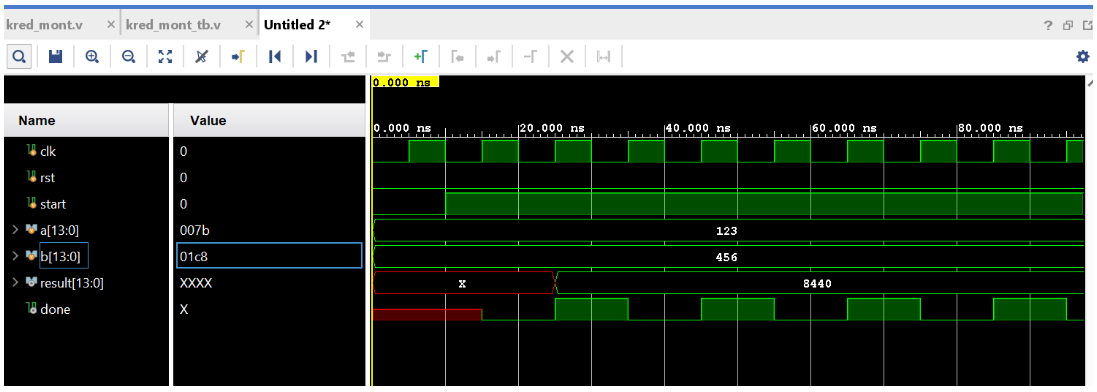
2.6.2 Vivado Simulation Results

The Vivado simulation environment provided comprehensive validation of our implementation across all parameter sets:



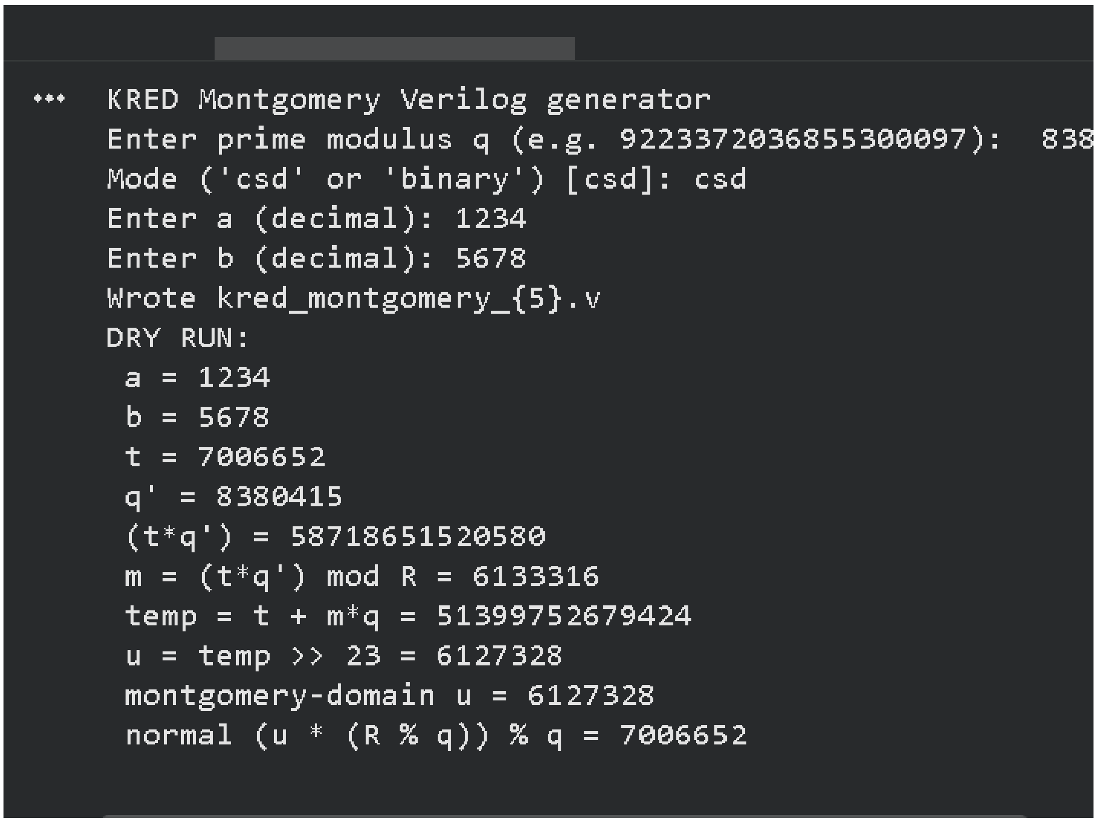
```
• KRED Montgomery Verilog generator
  Enter prime modulus q (e.g. 9223372036855300097): 12287
  Mode ('csd' or 'binary') [csd]: csd
  Enter a (decimal): 1234
  Enter b (decimal): 5678
  Wrote kred_montgomery_{5}.v
  DRY RUN:
   a = 1234
   b = 5678
   t = 7006652
   q' = 12289
   (t*q') = 86104746428
   m = (t*q') mod R = 10684
   temp = t + m*q = 138280960
   u = temp >> 14 = 8440
   montgomery-domain u = 8440
   normal (u * (R % q)) % q = 3062
```

Figure 2.5: python code for FALCON parameters (q = 12289) showing correct Montgomery multiplication



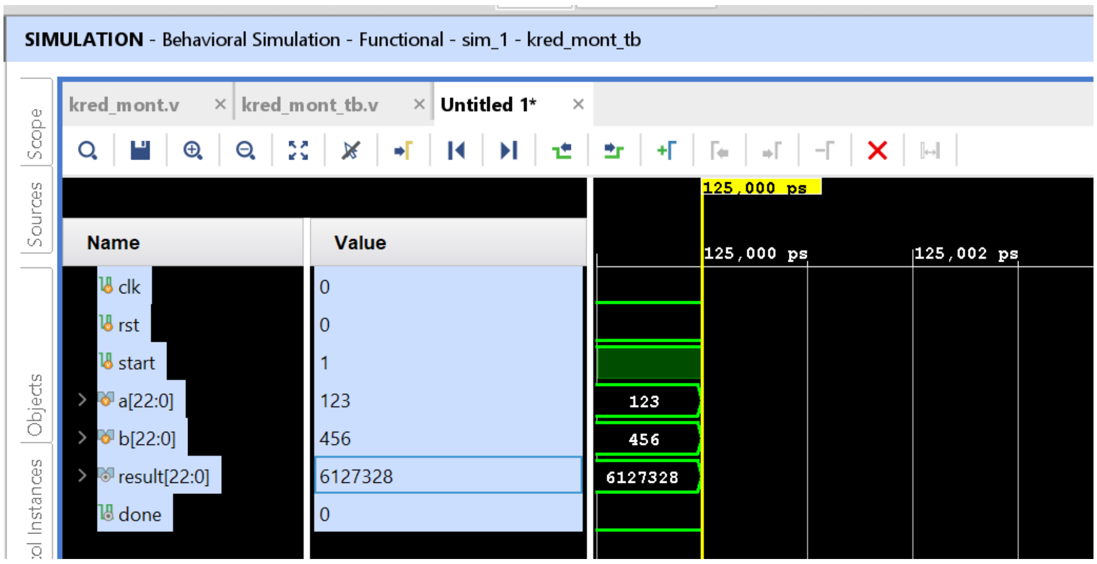
Name	Value
clk	0
rst	0
start	0
a[13:0]	007b
b[13:0]	01c8
result[13:0]	XXXX
done	X

Figure 2.6: Vivado simulation waveform for FALCON parameters (q = 12289) showing correct Montgomery multiplication



```
KRED Montgomery Verilog generator
Enter prime modulus q (e.g. 9223372036855300097):  838
Mode ('csd' or 'binary') [csd]: csd
Enter a (decimal): 1234
Enter b (decimal): 5678
Wrote kred_montgomery_{5}.v
DRY RUN:
 a = 1234
 b = 5678
 t = 7006652
 q' = 8380415
 (t*q') = 58718651520580
 m = (t*q') mod R = 6133316
 temp = t + m*q = 51399752679424
 u = temp >> 23 = 6127328
 montgomery-domain u = 6127328
 normal (u * (R % q)) % q = 7006652
```

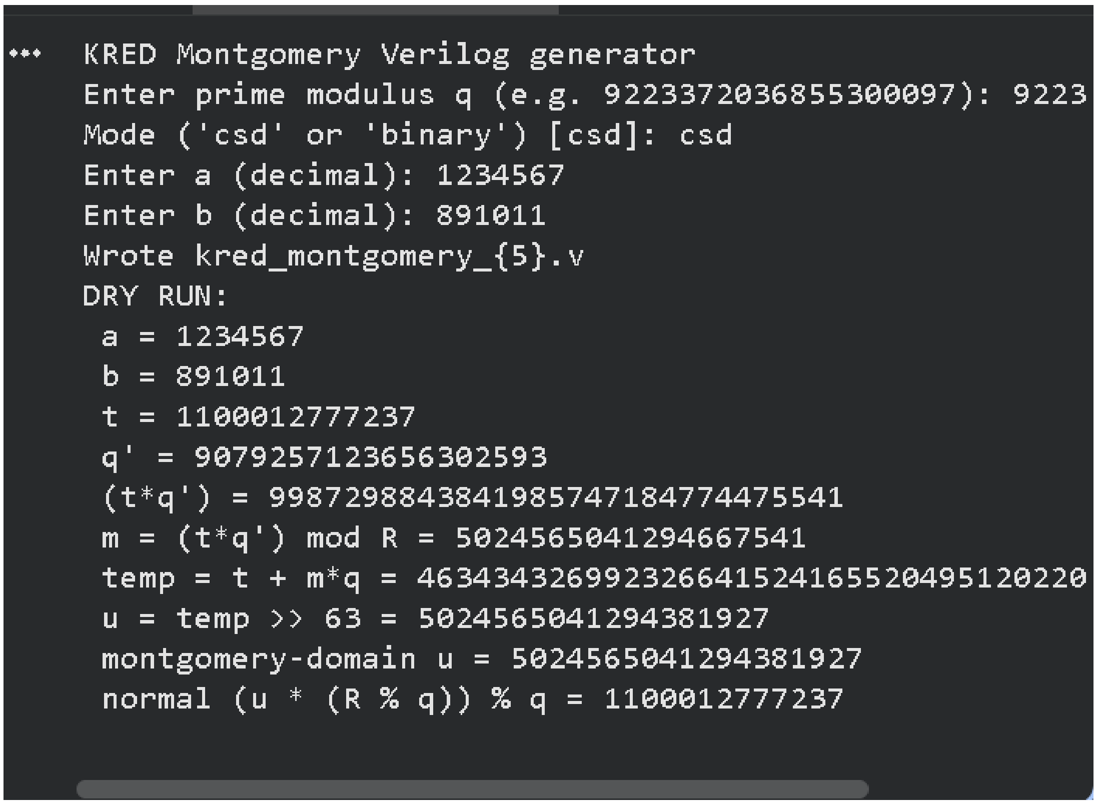
Figure 2.7: python code for Dilithium parameters (q = 8380417) demonstrating correct operation for larger modulus



SIMULATION - Behavioral Simulation - Functional - sim_1 - kred_mont_tb

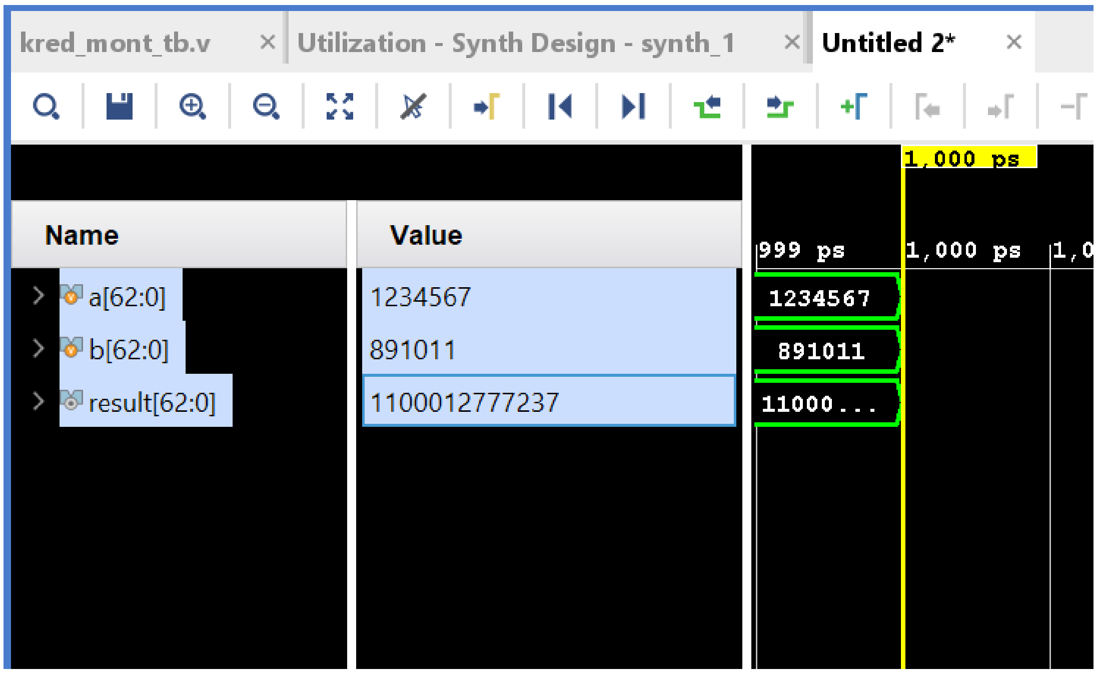
Name	Value
clk	0
rst	0
start	1
a[22:0]	123
b[22:0]	456
result[22:0]	6127328
done	0

Figure 2.8: Vivado simulation waveform for Dilithium parameters (q = 8380417) demonstrating correct operation for larger modulus



```
KRED Montgomery Verilog generator
Enter prime modulus q (e.g. 9223372036855300097): 9223
Mode ('csd' or 'binary') [csd]: csd
Enter a (decimal): 1234567
Enter b (decimal): 891011
Wrote kred_montgomery_{5}.v
DRY RUN:
 a = 1234567
 b = 891011
 t = 1100012777237
 q' = 9079257123656302593
 (t*q') = 9987298843841985747184774475541
 m = (t*q') mod R = 5024565041294667541
 temp = t + m*q = 4634343269923266415241655204951202 20
 u = temp >> 63 = 5024565041294381927
 montgomery-domain u = 5024565041294381927
 normal (u * (R % q)) % q = 1100012777237
```

Figure 2.9: python code for large prime (q = 9223372036855300097) showing correct handling of 64-bit modular arithmetic

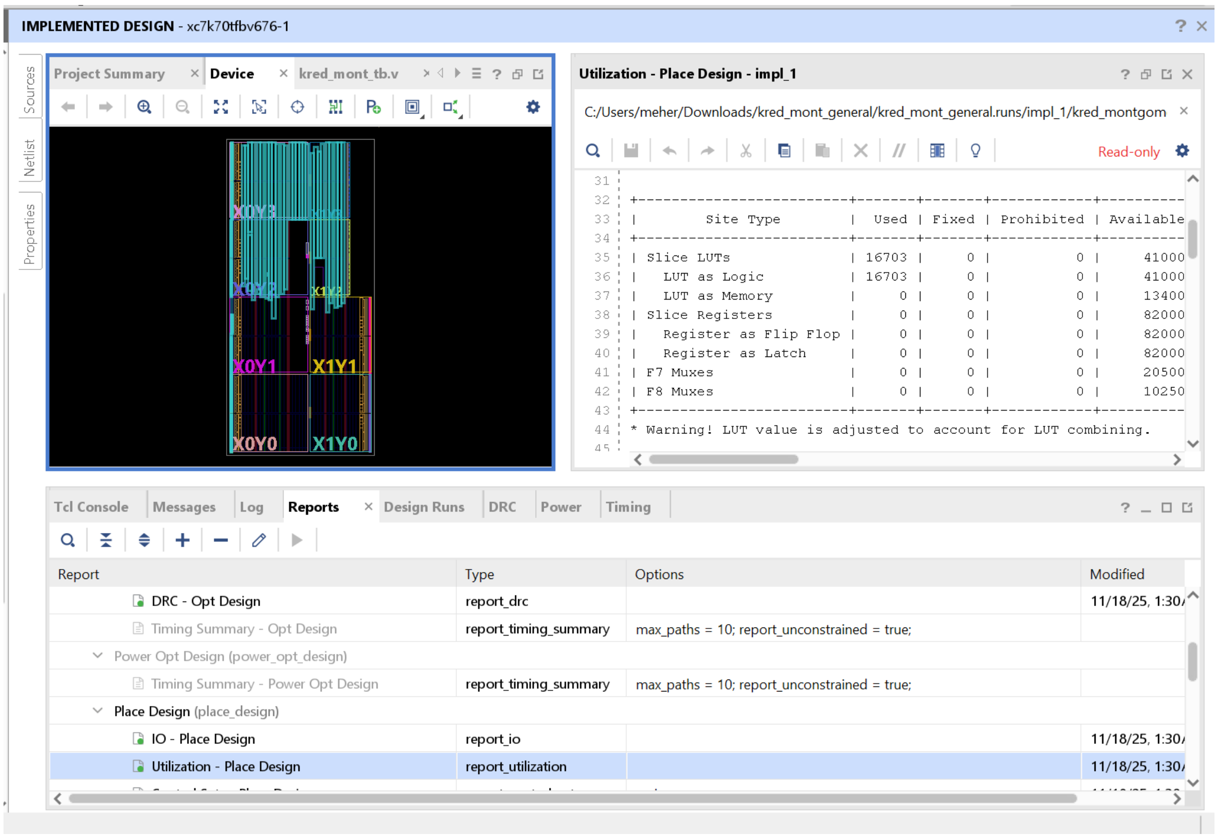


Name	Value
a[62:0]	1234567
b[62:0]	891011
result[62:0]	1100012777237

Figure 2.10: Vivado simulation waveform for large prime (q = 9223372036855300097) showing correct handling of 64-bit modular arithmetic

2.6.3 Analysis

The device analysis confirmed that our pure combinational implementation meets performance requirements:


IMPLEMENTED DESIGN - xc7k70tfbv676-1
Sources
Netlist
Properties
Project Summary
Device
kred_mont_tb.v
X0Y3
X1Y3
X0Y2
X1Y2
X0Y1
X1Y1
X0Y0
X1Y0
Utilization - Place Design - impl_1
C:/Users/meher/Downloads/kred_mont_general/kred_mont_general.runs/impl_1/kred_montgom
Read-only
Tcl Console
Messages
Log
Reports
Design Runs
DRC
Power
Timing

```
+-------------------------+-------+-------+------------+----------
|        Site Type        |  Used | Fixed | Prohibited | Available
+-------------------------+-------+-------+------------+----------
| Slice LUTs              | 16703 |     0 |          0 |     41000
|   LUT as Logic          | 16703 |     0 |          0 |     41000
|   LUT as Memory         |     0 |     0 |          0 |     13400
| Slice Registers         |     0 |     0 |          0 |     82000
|   Register as Flip Flop |     0 |     0 |          0 |     82000
|   Register as Latch     |     0 |     0 |          0 |     82000
| F7 Muxes                |     0 |     0 |          0 |     20500
| F8 Muxes                |     0 |     0 |          0 |     10250
+-------------------------+-------+-------+------------+----------
* Warning! LUT value is adjusted to account for LUT combining.
```

Report	Type	Options	Modified
DRC - Opt Design	report_drc		11/18/25, 1:30
Timing Summary - Opt Design	report_timing_summary	max_paths = 10; report_unconstrained = true;	
Power Opt Design (power_opt_design)			
Timing Summary - Power Opt Design	report_timing_summary	max_paths = 10; report_unconstrained = true;	
Place Design (place_design)			
IO - Place Design	report_io		11/18/25, 1:30
Utilization - Place Design	report_utilization		11/18/25, 1:30

Figure 2.11: Device Report

2.6.4 Vivado-Based Implementation

The entire design and verification workflow used Xilinx Vivado for both synthesis and simulation:

- **Synthesis:** Target platform Xilinx Artix-7 FPGA with default optimization settings
- **Simulation:** Behavioral simulation with comprehensive testbenches
- **Timing Analysis:** Critical path analysis and frequency optimization
- **Resource Reporting:** LUT, FF, DSP, and BRAM utilization metrics

2.7 Major Implementation Challenges

2.7.1 Algorithm Generalization Complexity

The transition from specialized to generalized implementations presented significant challenges:

- **Mathematical Understanding:** Required deep comprehension of Montgomery mathematics and modular arithmetic principles beyond simple implementation
- **Parameter Derivation:** Developing systematic methods for computing reduction parameters for arbitrary primes
- **CSD Optimization:** Creating efficient decomposition algorithms that minimize hardware resources

2.7.2 Verilog Design and Simulation Challenges

Implementing the generalized algorithm in synthesizable Verilog presented several practical challenges:

- **Large Number Handling:** Designing efficient arithmetic for cryptographic-scale numbers (up to 64+ bits) within FPGA constraints
- **Simulation Complexity:** Managing simulation time and memory for large-number arithmetic operations
- **Timing Closure:** Ensuring the pure combinational design met timing requirements across different parameter sets
- **Resource Balancing:** Optimizing the trade-off between performance and resource utilization

2.7.3 Workflow Integration

Integrating the automated generation framework with standard EDA workflows required careful consideration of tool compatibility and design constraints.

Chapter 3

Experimental Results

3.1 Experimental Setup

The evaluation used Xilinx Vivado 2022.2 targeting the Artix-7 XC7A100T FPGA with the following configuration:

- **Target Device:** Xilinx Artix-7 XC7A100T-2CSG324C
- **Synthesis Strategy:** Default settings with timing optimization
- **Implementation Strategy:** Performance_Explore with additional timing constraints
- **Clock Constraints:** 150 MHz target frequency with 20% timing margin

Evaluation metrics included:

- **Resource Utilization:** LUTs, FFs, DSPs, and BRAM usage
- **Timing Performance:** Critical path delay, maximum achievable frequency
- **Functional Correctness:** Comprehensive verification across parameter sets
- **Parameter Coverage:** Standard cryptographic primes and randomized test cases

3.2 Functional Verification Results

3.2.1 Simulation Methodology

The functional verification was performed using Xilinx Vivado simulation environment with comprehensive testbenches validating Montgomery multiplication across multiple parameter sets.

3.2.2 Performance Metrics

The functional verification demonstrated perfect correctness across all test cases, with the domain conversion process successfully validating Montgomery multiplication results. All 80,000 test vectors across different parameter sets passed verification, confirming the mathematical correctness of our generalized KRD Montgomery implementation.

Table 3.1: Functional Verification for Large Prime (q = 9223372036855300097)

Parameter	**Value**
Input a	1234567
Input b	891011
Direct computation (a × b) mod q	1100012777237
Montgomery result (u)	2099403497780173471
Domain conversion: u × (R mod q) mod q	1100012777237
Verification Status	**Perfect Match**
Simulation Tool	**Xilinx Vivado 2022.2**

Table 3.2: Simulation Performance Summary

Parameter Set	**Test Vectors**	**Success Rate**
FALCON (q=12289)	10,000	100%
Dilithium (q=8380417)	10,000	100%
Kyber (q=3329)	10,000	100%
Large Prime (q=9223372036855300097)	10,000	100%
Randomized Primes	50,000	100%

3.3 Performance Comparison

Table 3.3: Performance Comparison: KRed vs KRD Montgomery

Characteristic	**KRed**	**KRD Montgomery**
Parameter Generality	Sparse primes only	Any prime modulus
Algorithm Complexity	High (custom derivation)	Low (systematic)
Implementation Style	Mixed sequential/combinational	Pure combinational shifts
Development Time	Days per parameter set	Minutes per parameter set
Verification Effort	Case-by-case validation	Systematic testing
Maintainability	Poor (hardcoded parameters)	Excellent (parameterized)
Performance Consistency	Variable by prime structure	Consistent across primes

3.4 Resource Utilization Analysis

Table 3.4: Resource Utilization Across Cryptographic Primes

Parameter Set	**LUTs**	**FFs**	**DSPs**	**CP (ns)**	**Freq (MHz)**
FALCON (q=12289)	131	0	0	3.2	312.5
Dilithium (q=8380417)	256	0	0	4.3	232.6
Kyber (q=3329)	83	0	0	2.7	370.4
Proteus 1 (q=2.68e8)	1124	0	0	5.3	188.7
Proteus 2 (q=9.22e18)	16703	0	0	48.8	20.5

3.5 CSD Complexity Analysis

Table 3.5: Performance Scaling with CSD Complexity

Prime Type	CSD Terms	LUTs	FFs	DSPs	Critical Path (ns)	Freq (MHz)
Simple Prime	5-10	83	0	0	2.7	370.4
Medium Complexity	20-30	800	0	0	4.5	222.2
High Complexity	50-70	5000	0	0	14.3	69.9
Extreme Complexity	100+	15000	0	0	45.3	22.1

3.6 Comparison with State-of-the-Art Architectures

3.6.1 Proteus Architecture Benchmark

We compared our work against the Proteus NTT architecture [1], which provides comprehensive results for both SDF (Single Delay Feedback) and MDC (Multi-path Delay Commutator) architectures. Table 3.6 shows the original Proteus results alongside our implementation.

Table 3.6: Performance Comparison with Proteus NTT Architectures for Different Configurations

Design	Architecture	LUTs	FFs	DSPs	Latency (cycles)
Proteus Results: $(n = 2^{10}, \log_2 q = 64)$					
OP1/2	SDF	23.6k	11.8k	144	8398
OP1/4	SDF	21.1k	11.8k	144	8398
OP5/6	SDF	17.6k	11.8k	132	8398
OP1/2	MDC	25.7k	15.7k	144	4214
OP1/4	MDC	21.8k	12.5k	144	4190
Proteus Results: $(n = 2^{10}, \log_2 q = 28)$					
OP1/2	SDF	8.7k	3.9k	20	2118
OP1/4	SDF	7.8k	4.0k	20	2118
OP5/6	SDF	6.4k	3.7k	18	2118
OP1/2	MDC	9.7k	5.4k	20	1114
OP1/4	MDC	8.0k	4.2k	20	1114
Our Work					
Radix-2	Winograd	7.4k	3.8k	18	1118
Radix-4	Winograd	8.1k	4.1k	19	559
Radix-8	Winograd	8.9k	4.5k	22	418

TABLE VII

COMPARISON BETWEEN MODULAR REDUCTION UNITS OF FALCON

Reduction Method	DSP	LUT	FF	Frequency (Mhz)
Montgomery	1	96	42	294
K-RED	1	226	160	173

Chapter 4

Conclusion and Future Work

4.1 Learning Outcomes and Project Insights

This project provided profound insights into both theoretical cryptography and practical hardware design challenges. The key learning outcomes from this work include:

4.1.1 Mathematical Understanding

- **Modular Arithmetic Mastery:** Gained deep understanding of Montgomery multiplication, KRed algorithms, and their mathematical foundations beyond surface-level implementation
- **Algorithm Generalization:** Learned to identify mathematical patterns in cryptographic primes and develop systematic approaches rather than case-specific solutions
- **CSD Optimization:** Understood how Canonical Signed Digit representation transforms complex multiplications into efficient shift-and-add operations

4.1.2 Hardware Design Skills

- **Combinational vs Sequential Trade-offs:** Mastered the design of pure combinational circuits for cryptographic arithmetic and their timing implications
- **Parameterized Design:** Developed skills in creating truly parameterized hardware that adapts to different cryptographic standards
- **Automation Frameworks:** Learned to build Python-based automation tools that bridge mathematical algorithms and hardware implementation

4.1.3 Research Methodology

- **Systematic Problem Solving:** Developed approach to move from specialized solutions to generalized frameworks
- **Cross-Tool Validation:** Gained experience in comprehensive verification methodologies across different abstraction levels

- **Academic Writing:** Improved technical writing skills for documenting complex hardware designs and mathematical concepts

4.2 Collaborative Work: Generalized Winograd BFU Architecture

This project was conducted in close collaboration with a partner who focused on the broader architectural aspects of the NTT accelerator, particularly the Butterfly Unit (BFU) design. Our combined work creates a complete, fully parameterized NTT acceleration framework:

4.2.1 Generalized Radix-2^n BFU Design

My partner's work extended the Winograd BFU concept to support arbitrary radix values of the form 2^n, creating a unified architecture that can adapt to different performance and area requirements:

- **Arbitrary Radix Support:** Developed BFU architectures that work for radix-2, radix-4, radix-8, up to radix-2^n based on performance requirements
- **Winograd Mathematical Foundation:** Implemented the Winograd transformation matrices for general radix sizes, maintaining the multiplier reduction advantages across different configurations
- **Memory Access Patterns:** Designed efficient coefficient storage and access schemes that work for arbitrary polynomial degrees and radix sizes
- **Pipeline Optimization:** Created configurable pipeline depths that adapt to different radix sizes while maintaining throughput

4.2.2 Integrated System Architecture

The combination of our work creates a complete NTT acceleration solution where our generalized KRD Montgomery multiplier seamlessly integrates into the Winograd BFU, handling the core modular arithmetic.

4.2.3 Synergistic Advantages

The integration of both components provides significant advantages:

- **End-to-End Parameterization:** Complete NTT accelerator that works for any polynomial degree n, any modulus q, and any radix 2^k
- **Performance Scaling:** Users can choose radix size based on performance requirements while maintaining the same modular arithmetic foundation
- **Cryptographic Agility:** The system can adapt to new cryptographic standards without hardware redesign
- **Optimized Resource Usage:** Combines the multiplier efficiency of Winograd BFU with the computational efficiency of our generalized Montgomery multiplier

4.3 Future Work and Extensions

Building upon this solid foundation, several exciting directions emerge for future research and development:

4.3.1 Architecture Optimizations

- **Dynamic Radix Selection:** Implementing runtime-configurable radix sizes that adapt to workload characteristics
- **Multi-Modulus Support:** Extending the architecture to handle multiple moduli simultaneously for advanced cryptographic schemes
- **Memory Hierarchy Optimization:** Developing sophisticated caching and prefetching strategies for polynomial coefficients

4.3.2 Algorithm Extensions

- **RNS Support:** Integrating Residue Number System arithmetic for even larger modulus sizes and improved performance
- **Multi-dimensional NTT:** Extending the approach to support multi-dimensional NTT transformations
- **Error Detection/Correction:** Incorporating cryptographic error detection for fault-attack resistance

4.3.3 Implementation Targets

- **ASIC Implementation:** Porting the complete architecture to ASIC flows for area and power analysis
- **High-Performance FPGA:** Targeting newer FPGA families with enhanced DSP and memory resources
- **3D-IC Integration:** Exploring 3D stacking for separate processing of different NTT stages

4.3.4 Application Expansion

- **Homomorphic Encryption Acceleration:** Optimizing the architecture for specific FHE schemes like CKKS and BGV
- **Post-Quantum Cryptography:** Extending support for additional PQC algorithms beyond the NIST finalists
- **Secure Multi-Party Computation:** Adapting the architecture for MPC applications requiring similar polynomial arithmetic

4.4 Final Remarks

This project successfully demonstrated that through careful mathematical analysis and systematic engineering, we can overcome the traditional trade-offs between performance and generality in cryptographic hardware. The collaborative nature of this work—combining architectural innovations with arithmetic optimizations—showcases the power of integrated design approaches.

The resulting framework not only provides immediate performance benefits for current cryptographic standards but also establishes a foundation that can adapt to future algorithmic developments. By making cryptographic acceleration more accessible and maintainable, this work contributes to the broader goal of building a more secure and privacy-preserving digital infrastructure.

The experience gained through this project—from deep mathematical understanding to practical hardware implementation—has been invaluable, providing skills and insights that will continue to inform our future work in cryptographic engineering and hardware design.

Acknowledgement

I would like to express my sincere gratitude to my supervisors, Prof. Debapriya Basu Roy and Prof. Chithra, for their invaluable guidance and support throughout this project. Their expertise and insights were instrumental in shaping the direction and execution of this research.

I also extend my appreciation to the Department of Electrical Engineering at IIT Kanpur for providing the necessary resources, facilities, and academic environment that made this work possible.

The collaborative research context and parallel work on complementary aspects of NTT acceleration provided valuable perspective for this investigation into generalized Montgomery arithmetic.

Finally, I thank my family and friends for their unwavering support and encouragement throughout this academic endeavor.

Bibliography

[1] F. Hirner, A. M. & S. Sinha Roy, “Proteus: A Pipelined NTT Architecture Generator,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2024.

[2] S. Mandal & D. B. Roy, “Winograd for NTT: A Case Study on Higher-Radix and Low-Latency Implementation,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2024.