Achieving NTRU with Montgomery Multiplication

Colleen O'Rourke and Berk Sunar, *Member*, *IEEE*

Abstract—In this paper, we propose a new unified architecture that utilizes the Montgomery Multiplication algorithm to perform a modular multiplication for both integers and binary polynomials and NTRU's polynomial multiplications. The unified design is capable of supporting a majority of public-key cryptosystems such as NTRU, RSA, Diffie-Hellman key exchange, and Elliptic Curve schemes, among others. Furthermore, the architecture is highly efficient in terms of area and speed.

Index Terms—Cryptography, NTRU, unified architectures, Montgomery multipliers, Montgomery multiplication, finite fields.

1 Introduction

For a majority of public key schemes such as RSA [1], ECC [2], [3], and NTRU [4], most of the time is spent in performing modular multiplications. These multiplications occur as either integer modular multiplications, binary polynomial modular multiplications, or polynomial multiplications over an integer ring. Developing a *unified architecture* that can perform modular multiplications for all representations is attractive because it provides the following key features: algorithm agility, resource utilization, and compatibility.

Algorithm Agility allows the user to switch cryptographic algorithms during *runtime*. Therefore, if current cryptosystems become obsolete due to new security needs, several back-up cryptosystems can be used as a replacement without requiring redesign.

Resource Utilization uses the same hardware to perform the majority of the arithmetic used for the different supported algorithms. Therefore, it reduces power and area consumption, which is critical for resource constrained applications. Alternatively, the architecture can be designed so that each algorithm allocates a separate area, which is not ideal.

Compatibility allows interoperability of various applications through a multitude of cryptographic algorithms. The problem stems from the fact that high end platforms may utilize cryptosystems such as RSA, whereas, for smart cards or other embedded applications, ECC or NTRU may be more reasonable choices.

Since Montgomery Multiplication (MM) is capable of improving the performance of integer and binary polynomial modular multiplications [5], [6], a number of unified architectures have been proposed. The work in [7] introduces a scalable and unified multiplier. This design supports both types of arithmetic through the introduction of the "dual-field adder" which performs addition with and without carry propagation. A word-level bit-serial version of the MM algorithm is utilized to avoid designing a dual field multiplier. Scaling is achieved by increasing the number of fixed area units. In addition, the pipeline depth and word size can be configured to meet the desired area and performance specifications.

In [8], a unified scalable Montgomery Multiplier design is presented. The design is based on the word-level MM algorithm which is implemented using a dual field multiplier that supports $w \times w$-bit multiplications. The high-radix design utilizes the addition tree method to reduce the delay path to $\log_{\frac{3}{2}} w$ (instead of w) in the dual field multiplier and adder. The word size of the multiplier core and the pipelining depth can be configured before implementation.

In extension to the previous unified architectures, our design incorporates a third type of operation with the MM algorithm: the polynomial multiplication in NTRU's Public Key Cryptosystem, which is simply the usual convolution product of two vectors. By unifying NTRU's polynomial multiplication to the MM algorithm, a more flexible unified architecture can be designed to support more cryptosystems and a wider array of applications with a single processing element. To the best of our knowledge, no work has been published on the unification of NTRU with MM.

The rest of this paper is organized as follows: Section 2 provides a brief description of the NTRU Public Key Cryptosystem and Montgomery Multiplication. Section 3 presents the word-level Montgomery Multiplication algorithm and discusses the mathematical and algorithmic modifications that are necessary to provide support for NTRU. The new unified architecture is presented in Section 4. Section 5 describes how the unified multiplier can support every polynomial multiplication required by NTRU's major procedures. A performance analysis is presented in Section 6. Future improvements for the design are discussed in Section 7. Finally, conclusions are drawn in Section 8.

- *C. O'Rourke is with the Weapons System Design Lab, Raytheon Electronic Systems, 50 Apple Hill Rd., Tewksbury, MA 01876. E-mail: Colleen_M_O'Rourke@raytheon.com.*
- *B. Sunar is with the Worcester Polytechnic Institute, Atwater Kent Room 302, 100 Institute Rd., Worcester, MA 01609. E-mail: sunar@wpi.edu.*

Manuscript received 30 May 2002; revised 26 Nov. 2002; accepted 26 Nov. 2002.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 117866.

2 Preliminaries

This section introduces two topics to the reader as background for the sections that follow.

2.1 NTRU Public Key Crytosystem

NTRU is a polynomial ring-based public key cryptosystem that was fully introduced in 1998 [4]. The scheme is set up by three integers (N, p, q) such that:

- N is prime,
- p and q are relatively prime, $\gcd(p, q) = 1$, and
- q is much much larger than p.

NTRU is based on polynomial additions and multiplications in the ring $R = Z[x]/(x^N - 1)$. We use "$*$" to denote a polynomial multiplication in R, which is the cyclic convolution of two polynomials. After completion of a polynomial multiplication or addition, the coefficients of the resulting polynomial need to be reduced either modulo q or p. As a side note, the key creation process also requires two polynomial inversions, which can be computed using the Extended Euclidean Algorithm. NTRU requires approximately $\mathcal{O}(N^2)$ operations and a key length of $\mathcal{O}(N)$. More information on NTRU can be found in [4] and [9]. We briefly outline the procedures below.

Key Generation. To generate the public key, the user must:

- Choose a secret key, a random polynomial $f \in R$, where coefficients are in $(-\frac{p}{2}, \frac{p}{2})$,[1]
- Choose a random polynomial, $g \in R$, where coefficients are in $(-\frac{p}{2}, \frac{p}{2})$, and
- Compute the inverse polynomial F_q of the secret key f modulo q.

Once the above has been completed, the public key, h, is found as

$$h = F_q * g \pmod{q}.$$

Encryption. The encrypted message is computed as

$$e = pr * h + m \pmod{q},$$

where the message, $m \in R$, and the random polynomial, $r \in R$, have coefficients reduced modulo p.

Decryption. The decryption procedure requires three steps:

- $a = f * e \pmod{q}$
- shift coefficients of a to the range $(-\frac{q}{2}, \frac{q}{2})$, and
- $d = F_p * a \pmod{p}$.

The last step of decryption requires the user to compute the inverse polynomial F_p of the secret key f modulo p. The decryption process outlined above will recover the original message ($d = m$).

Note that, by choosing $f = 1 + p \cdot f_1$, where $f_1 \in R$ in the key generation step, the second polynomial multiplication in the decryption step is eliminated since $F_p = 1 \pmod{p}$.

1. For $p = 2$ coefficients are in $\{0, 1\}$.

2.2 Montgomery Multiplication

MM was introduced by P.L. Montgomery in [5] to improve the performance of computing the modular multiplication

$$c = a \cdot b \pmod{m},$$

where a, b, c, and m are integers. The reduction modulo m requires a division which is a costly operation. Instead, Montgomery replaces this division with a series of inexpensive shift operations. Montgomery's technique utilizes a residue representation:

$$\begin{aligned} \overline{a} &= a \cdot R \pmod{m} \\ \overline{b} &= b \cdot R \pmod{m}. \end{aligned} \tag{1}$$

There are two restrictions on choosing an integer residue, R:

- $R > m$ and
- $\gcd(R, m) = 1$.

After computation of the residues, the MM algorithm computes the product, c:

$$MM(\overline{a}, \overline{b}) = \overline{a} \cdot \overline{b} \cdot R^{-1} = c \cdot R = \overline{c} \pmod{m}.$$

As indicated by the bar notation above, the result, c, is in residue form. To convert the result back to nonresidue format, one more MM is computed:

$$MM(\overline{c}, 1) = \overline{c} \cdot 1 \cdot R^{-1} = c \pmod{m}.$$

MM can also perform modular multiplications of binary polynomials, as explained in [6].

Ideally, Montgomery's technique should be used for applications that require multiple modular multiplications over the same modulus (e.g., modular exponentiations) since the residue is preserved for each consecutive multiplication.

3 Our Contribution

In this section, we describe the mathematical and algorithmic changes that are necessary to achieve NTRU's polynomial multiplication using MM. As a note, we represent all operands as bit strings, which are, according to the context, interpreted as either one of the following:

- coefficients of the expansion of an integer in powers of 2,
- coefficients of the expansion of a binary polynomial in powers of x,
- the binary representation of the integer coefficients of a polynomial defined over an integer ring.

There are differences in the way the three types of multiplications are computed. In integer multiplications, carries are propagated through the entire bit string representation of the product. In binary polynomial multiplications, however, no carries are propagated. In NTRU's polynomial multiplications, carries are propagated only within the integer coefficients but not among coefficents. These differences are minor and, in a hardware implementation, they can be implemented by inhibiting carry propagation when necessary. While regular modular multiplications can be unified using this method, to unify MM

with NTRU, additional changes are necessary, which are described below.

3.1 Achieving NTRU with Montgomery Multiplication

To achieve NTRU using MM, the extra R factor has to be taken into consideration. We observe that, by setting the residue $R = x^N$ and the modulus $m = x^N - 1$ in MM, the following property is introduced:

$$R = x^N \equiv 1 \pmod{x^N - 1}.$$

When this property is applied to Montgomery's method, the residue of an operand is simply the operand itself:

$$\overline{b} = b \cdot R = b \cdot 1 = b \pmod{x^N - 1}$$

and the Montgomery product is in nonresidue format:

$$\begin{aligned} MM(\overline{a}, \overline{b}) &= a \cdot b \cdot R^{-1} \pmod{x^N - 1} \\ &= c \cdot (1)^{-1} = c \pmod{x^N - 1}. \end{aligned}$$

Therefore, by restricting $R = x^N$, NTRU's operands never need to be converted to and from residue form to receive the correct result from MM.

As a result of the unification, setting $R = x^N$ restricts the residue and modulus for the integer and binary polynomial cases as follows:
For integers:

$$\begin{aligned} &R = 2^{N \cdot w} \text{ and} \\ &m < 2^{N \cdot w}. \end{aligned}$$

For binary polynomials:

$$\begin{aligned} R(x) &= x^{N \cdot w} \quad \text{and} \\ \text{degree}(m(x)) &\leq N \cdot w. \end{aligned}$$

3.2 Unified Word-Level Algorithm

The word-level integer MM algorithm is given in Algorithm 3.2, which is a slightly modified version of the FIOS method introduced in [10]. In this algorithm,

- $a[i]$, $b[j]$, $c[i]$, and $m[i]$ represent individual elements of the word array representations of integers a, b, c, and m, respectively,
- the least significant word is stored in the 0th element of the array (e.g., $m[0]$),
- the word size is w bits in length,
- n is the number of words in the arrays,
- $m[0]' = -m[0]^{-1} \bmod 2^w$ is precomputed, and
- CS has a total length of $2^w + 1$ bits such that:
  - C, the most significant word, is $w + 1$ bits long and
  - S, the least significant word, is w bits long.

After completion of Algorithm 3.2, a long subtraction of $c = c - m$ may be necessary if $c \geq m$.

Word-Level Montgomery Algorithm for $GF(p)$
Step 1: for $j = 0$ to $n - 1$
Step 2: $CS = (a[0] \cdot b[j]) + c[0]$
Step 3: $U = S \cdot m[0]' \bmod 2^w$
Step 4: $CS = CS + (m[0] \cdot U)$
Step 5: $CS \gg w$
Step 6: for $i = 1$ to $n - 1$
Step 7: $CS = CS + (a[i] \cdot b[j]) + (m[i] \cdot U) + c[i]$
Step 8: $c[i - 1] = S$
Step 9: $CS \gg w$
Step 10: endfor
Step 11: $c[n - 1] = S$
Step 12: endfor

With a few modifications, as proposed in [6], Algorithm 3.2 can be used for binary polynomials as well. The modifications include changing integer multiplications to polynomial multiplications and additions to word size XOR operations. Also, since $GF(2)$ arithmetic eliminates carry propagation, CS only needs to be $2w$ bits long and the long subtraction after completion of the algorithm is no longer necessary.

We now describe the algorithmic changes that are necessary to achieve NTRU's polynomial multiplication with the MM algorithm. NTRU's operands are stored as word arrays where each coefficient is placed in a w-bit word. Since NTRU's modulus is comprised of $N + 1$ words (coefficients), the inner loop in Algorithm 3.2 needs to be incremented by one to process all words of the modulus. In addition, the parameter n is changed to NTRU's parameter N. We assume that the operands and moduli for all three cases are stored as $N + 1$ word arrays. The new algorithm is shown in Algorithm 3.2.

Word-Level Montgomery Algorithm for $GF(p)$, $GF(2^k)$, and NTRU
Step 1: for $j = 0$ to $N - 1$
Step 2: $CS = (a[0] \cdot b[j]) + c[0]$
Step 3: $U = S \cdot m[0]' \bmod 2^w$
Step 4: $CS = CS + (m[0] \cdot U)$
// If NTRU, reduce mod 2^w
Step 5: $CS \gg w$
Step 6: for $i = 1$ to N
Step 7: $CS = CS + (a[i] \cdot b[j]) + (m[i] \cdot U) + c[i]$
// If NTRU, reduce mod 2^w
Step 8: $c[i - 1] = S$
Step 9: $CS \gg w$
Step 10: endfor
Step 11: $c[N] = S$
Step 12: endfor

There is one additional change that is necessary to make NTRU's polynomial multiplication work with the MM algorithm. In Steps 4 and 7 of Algorithm 3.2, the result needs to be reduced modulo q. In NTRU implementations, q is selected as 2^w for more efficient reductions. For this case, there is no carry propagation from one word to the next. Hence, the upper word of the result from Steps 4 and Step 7 needs to be cleared before it is reprocessed in Step 7.

4 Hardware Design

In this section, we detail the hardware architecture that utilizes the MM algorithm to perform integer modular multiplications, binary polynomial modular multiplications,

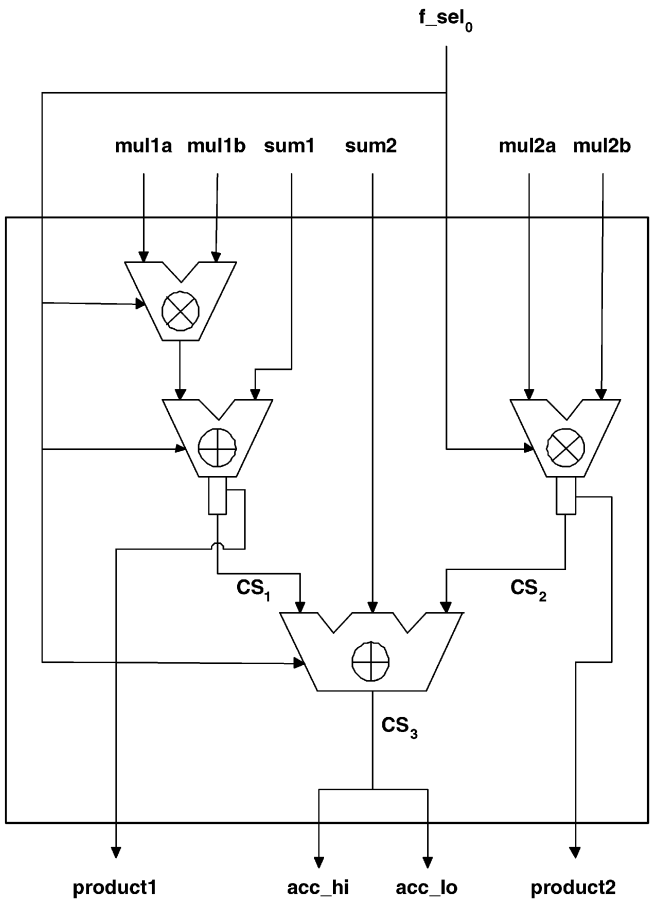


Fig. 1. The Montgomery Multiplier Core.

and NTRU's polynomial multiplications. Since NTRU's operands and modulus are polynomials whose coefficients are w-bit integers, a high-radix word-level design is the most convenient. We start to build a new unified core by extending a high-radix word-level Montgomery Multiplier core, which implements Algorithm 3.2, to support NTRU with minimal change to the hardware. To simplify the control logic, the Montgomery Multiplier core is fixed for a word size $w = 8$ bits to support the maximum length of NTRU's polynomial coefficients. The following sections first present the Montgomery Multiplier core, introduce the new unified core, then discuss the control system which orchestrates the operations of Algorithm 3.2.

4.1 High-Radix Montgomery Multiplier Core

A slightly modified version of the high-radix Montgomery Multiplier Core in [8] is shown in Fig. 1. The core is capable of performing all operations required by Algorithm 3.2 to achieve integer and binary polynomial modular multiplications. The core consists of two 8×8-bit dual field multipliers, an 8×8-bit dual field adder, and a three-way dual field adder. The three-way dual field adder accepts two operands that are 16 bits in length and a third operand that is 9 bits in length. The $\mathbf{f\_sel_0}$ determines whether the dual field components will perform integer or binary polynomial arithmetic. If $\mathbf{f\_sel_0} = 1$, the components perform integer arithmetic. Otherwise, the components perform binary polynomial arithmetic. The following describes how each step of Algorithm 3.2 is supported by the core.

- **Step 2:** $\mathbf{CS} = (\mathbf{a}[\mathbf{0}] \cdot \mathbf{b}[\mathbf{j}]) + \mathbf{c}[\mathbf{0}]$
  The first multiplier and adder (left side of Fig. 1) are utilized for Step 2.
- **Step 3:** $\mathbf{U} = \mathbf{S} \cdot \mathbf{m}[\mathbf{0}]' \bmod \mathbf{2^w}$
  For Step 3, the second multiplier is utilized because the only operation required is a multiplication. The S in Step 3 is the lower word ($product1$) of the result, CS_1, of Step 2. Also, the equation requires the reduction modulo 2^w, which is simply the extraction of the lower word ($product2$) of CS_2. Since $product2$ will need to be accessible for each execution of Step 7, $prodcut2$ is stored to register U outside of the core.
- **Step 4:** $\mathbf{CS} = \mathbf{CS} + (\mathbf{m}[\mathbf{0}] \cdot \mathbf{U})$
  The operation in Step 4 requires an 8×8-bit multiplication then a 16×16-bit addition. As a result, the second multiplier and the three-way adder are used for the computation of Step 4. The result from the operation in Step 2 (CS_1), the result of $m[0] \cdot U$ (CS_2), and $sum2 = 0$ are passed on to the three-way adder. After processing the three inputs, the three-way adder produces the final result for Step 4, CS_3.
- **Step 5 or 9:** $\mathbf{CS} = \mathbf{CS} \gg \mathbf{w}$ **and Step 7:** $\mathbf{CS} = \mathbf{CS} + (\mathbf{a}[\mathbf{i}] \cdot \mathbf{b}[\mathbf{j}]) + (\mathbf{m}[\mathbf{i}] \cdot \mathbf{U}) + \mathbf{c}[\mathbf{i}]$
  For these steps, all of the components of the core are used. The first multiplier and adder combination computes $(a[i] \cdot b[j]) + c[i]$ to produce the intermediate result, CS_1. The second multiplier computes the second multiplication ($m[i] \cdot U$) in parallel with the first multiplier to produce the second intermediate result, CS_2. Finally, the three-way adder adds the two intermediate results, CS_1 and CS_2, and $sum2$ to produce the final result, CS_3. For clarification, $sum2$ receives the shifted result from either Step 5 (when entering the loop) or Step 9 (when within the loop), which is just the upper 9 bits, acc_hi, of previous operation in Step 4 or Step 7, respectively.
- **Step 8:** $\mathbf{c}[\mathbf{i} - \mathbf{1}] = \mathbf{S}$
  This step simply assigns the lower word of the result from Step 7, acc_lo, to the respective location in the word vector, c, which stores the final integer result.
- **Step 11:** $\mathbf{c}[\mathbf{n} - \mathbf{1}] = \mathbf{S}$
  This step requires that the lower word of the shift operation in Step 9 be assigned to the respective location in the word vector, c. Since the lower word, S, is 8 bits, only the least significant 8 bits of acc_hi are assigned to the output.

4.2 Unified Core

The Montgomery Multiplier core is extended to a unified core, which is able to support all operations in Algorithm 3.2. The modifications mentioned in Section 3.2 do not require any hardware modifications to the Montgomery Multiplier core. Instead, these modifications require either a simple change to the control logic or additional hardware outside the Montgomery Multiplier

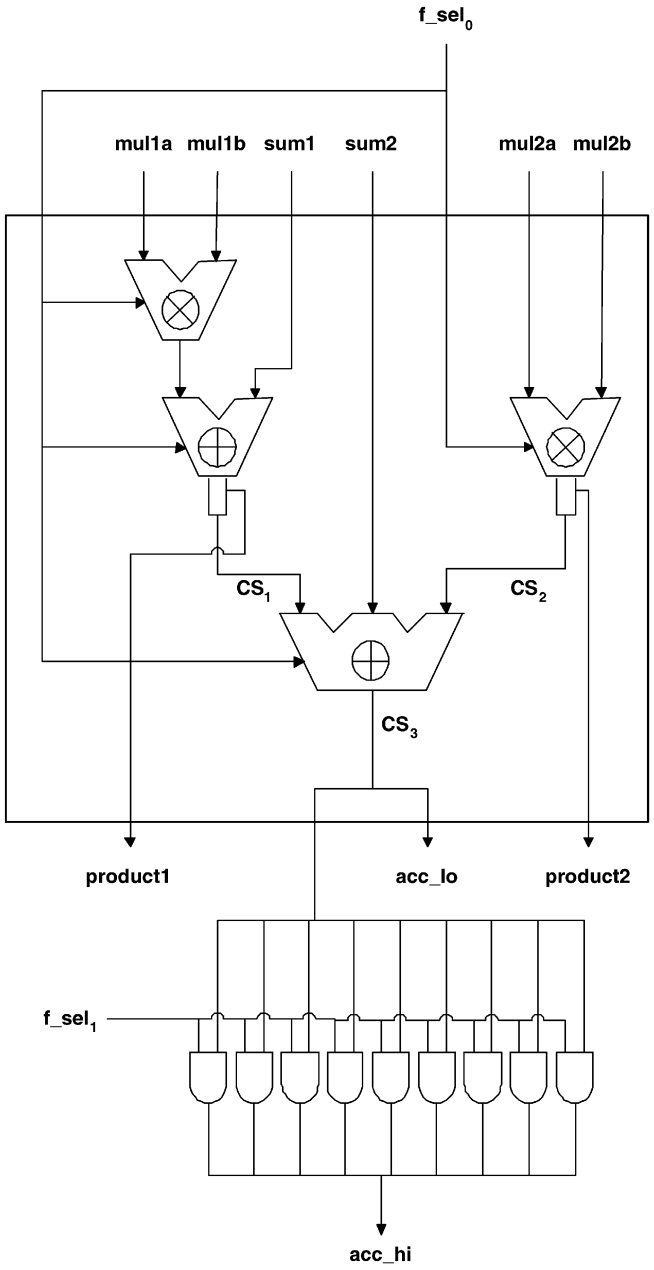


Fig. 2. NTRU and Montgomery Multiplier.

core to create the new unified core, as shown in Fig. 2. As a note, the design assumes that NTRU's integer moduli are $p < q$ and $q = 2^k$, where $k \leq 8$. In addition, the new unified architecture reduces the coefficients of NTRU's product polynomial modulo q.

The first modification mentioned earlier requires that the length of the inner loop in Algorithm 3.2 be incremented by one as shown in Algorithm 3.2. Although this modification requires the control logic to perform an additional iteration, this does not necessarily mean that the counter size has to change. For instance, changing the control to count from 500 to 501 still requires a 9-bit counter. With the exception of $N + 1$ being a power of 2 (e.g., incrementing the count from 511 to 512), no additional hardware on top of the original control (with no NTRU support) is required to perform this additional iteration.

TABLE 1
Assignment of the f_sel Signal

f_sel	function
00	Nothing
01	NTRU
10	$GF(2^k)$
11	$GF(p)$

The next modification requires that the "carry word" from the results of Step 4 and Step 7 be cleared when NTRU is selected. This "carry word" is the upper word of the result from the three-way adder, acc_hi. In order to distinguish between the functions, the unified architecture supports a two-bit $\mathbf{f\_sel}$ signal is necessary. The least significant bit, $\mathbf{f\_sel_0}$, determines whether the multipliers and adders perform integer multiplications and additions ($\mathbf{f\_sel_0} = 1$) or polynomial multiplications and additions ($\mathbf{f\_sel_0} = 0$). The most significant bit, $\mathbf{f\_sel_1}$, determines whether NTRU is selected. Since NTRU's polynomial multiplication relies on the integer multiplication and addition of its coefficients, $\mathbf{f\_sel_0}$ needs to be set to 1. Refer to Table 1 for clarification on the assignment of the $\mathbf{f\_sel}$ signal and its selected function. The "carry word," acc_hi, is cleared by AND-ing each bit with $\mathbf{f\_sel_1}$, as shown in Fig. 2. If NTRU is selected, $\mathbf{f\_sel_1} = 0$, then the AND gates zero out all 9 bits of acc_hi as needed. Otherwise, acc_hi passes through the AND gates unchanged. Therefore, NTRU can be supported using the Montgomery Multiplier core with the addition of only nine gates.

4.3 Control

The control assumes that the word arrays a, b, c, and m reside in separate memory caches. It is also assumed that $m[0]'$ is precomputed and stored in a register. For the NTRU case, $m[0]'$ is always set to 1. By using the Montgomery Multiplier core and the additional hardware shown in Fig. 2, the control executes Algorithm 3.2 in seven stages, as shown in Fig. 3. The host system initializes the unified architecture for processing, by asserting the *reset* signal. At this point, Algorithm 3.2 is set up by the control in Stage 0. Then, the outer loop in Algorithm 3.2 is performed by executing Stages 1 through 6 N times. Finally, the inner loop in Algorithm 3.2 is performed by executing Stages 4 through 5 N times. The stages of the implementation of Algorithm 3.2 and their associated operations are explained in detail below.

- **Stage 0** initializes the indexes i and j to zero and transmits the addresses for $a[i]$, $b[j]$, and $c[i]$ to the respective caches so that the data will be available for the next stage.
- **Stage 1** is responsible for setting up the core to perform the operation in Step 2. Since Stage 2 does not require any new data from memory and $m[0]'$ is precomputed, there is no need to transmit any new addresses to memory.
- **Stage 2** is responsible for setting up the core to perform the operation in Step 3. In addition, this

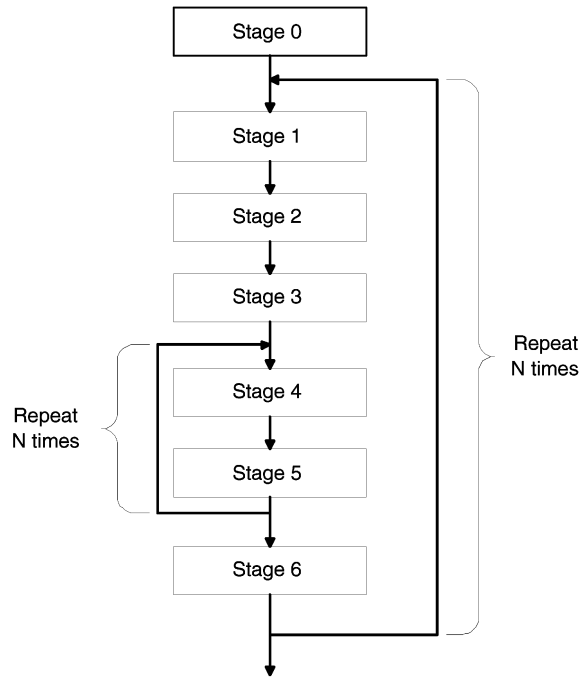


Fig. 3. Control sequence of Algorithm 3.2.

stage transmits the address of $m[0]$ to memory so it will be available for processing in the next stage.

- **Stage 3** is responsible for setting up the core to perform the operation in Step 4. In addition, this stage increments the index i and transmits the new addresses for a[i], b[j], m[i], and c[i] so that they will be available for Stage 4.
- **Stage 4** is responsible for setting up the core to perform the operation in Step 7. Also, this stage prepares for the write operation in Stage 5 by transmitting the address for $c[i-1]$.
- **Stage 5** writes the lower word of CS_3 to the respective word location of c (Step 8). Also, this stage does either one of the following: If the $for\ loop$ has not completed, then it increments the index i and transmits the addresses for $a[i]$, $b[j]$, $c[i]$, and $m[i]$ in preparation for Stage 4. Otherwise, only the address for $c[i]$ is tranmitted for Stage 6.
- **Stage 6** writes the lower 8 bits of acc_hi to the Nth element in the word array c (Step 11). Also, this stage prepares for the reexecution of the outer loop by incrementing the index j, clearing the index i, and transmitting the addresses for a[i], b[j], c[i].

5 Supported Operations and Limitations for NTRU

Previous works [7], [8] have established how the Montgomery Multiplier can achieve integer and binary polynomial modular multiplications. However, this is the first realization of a unified architecture that also supports NTRU's polynomial multiplication. This section details how the unified multiplier provides support for all the polynomial multiplications required by the key creation and encryption procedures and the decryption procedure with the addition of a modulo p reduction circuit. As mentioned earlier, we restrict the integer moduli to $p < q$ and $q = 2^k$, where $k \leq 8$. The polynomial size N, however, can be set to arbitrary lengths and is only limited by the size of the counters in the control unit. The following indicates which operations are supported and the associated assumptions.

1. **Public Key Creation:** $h = F_q * g \pmod q$
   The unified multiplier can perform the full operation above assuming that:
   - The random polynomial, g, has coefficients from $\{0, 1, 2^k - 1\}$ [2] and
   - The inverse polynomial F_q of the private key f modulo q has been precomputed and has coefficients in the range $[0, q-1]$.
2. **Encryption:** $e = pr * h + m \pmod q$
   The unified multiplier can only perform the multiplication of $r * h$ (mod q). It is assumed that:
   - The random polynomial, r, has coefficients from $\{0, 1, 2^k - 1\}$,
   - The message, m, is added outside of the multiplier and has coefficients from $\{0, 1, 2^k - 1\}$,
   - The integer multiplication of p occurs outside of the multiplier either after the multiplier has computed $r * h$ (mod q) or with the public key, h, prior to encryption.
3. **Decryption:**
   Originally, the decryption process for NTRU consists of three steps:
   a. $a = f * e \pmod q$,
   b. shift coefficients of a from $[0, q-1]$ to $[-\frac{q}{2}, \frac{q}{2}]$ and
   c. $d = F_p * a \pmod p$.
   The unified multiplier is able to compute Step a, while Step b needs to be performed outside of the multiplier. Step c cannot be computed by the multiplier because polynomial a's coefficients are no longer unsigned. In addition, the multiplier does not support reduction modulo p. Fortunately, there is a way to slightly modify the steps so that the unified multiplier can perform the polynomial multiplications within the decryption process with minimal additional hardware. Four steps are now required:
   a. $a = f * e \pmod q$
      The unified multiplier can perform the full operation above as long as the random polynomial, f, has coefficients from $\{0, 1, 2^k - 1\}$.
   b. Shift coefficients of a from $[0, q-1]$ $[-\frac{q}{2}, \frac{q}{2}]$ outside of the unified multiplier.
   c. $b = a \pmod p$
      The unified multiplier does not perform this operation. It is assumed that the user will reduce the coefficients of a modulo p. In the end, b will have coefficients from $\{0, 1, 2\}$.
   d. $d = F_p * b \pmod p$
      As a result of Steps b and c, the two polynomials, b and F_p, are now compatible for

2. Note that, $2^k - 1$ serves as $-1 \bmod q$.

TABLE 2
Performance Analysis for Unified Design

N	1	20	100	128	200	300	400	500	600
# gates for MM core	2054	2054	2054	2054	2054	2054	2054	2054	2054
# gates for control	440	636	761	817	809	868	864	878	911
# gates for NTRU support	9	9	19	9	9	9	9	9	9
Total # of gates	2504	2700	2825	2881	2873	2932	2928	2942	2975
% core	82.0%	76.1%	72.7%	71.3%	71.5%	70.1%	70.2%	69.8%	69.0%
% control	17.6%	23.6%	26.9%	28.4%	28.2%	29.6%	29.5%	29.8%	30.6%
% support for NTRU	0.4%	0.4%	0.4%	0.3%	0.3%	0.3%	0.3%	0.3%	0.3%
Frequency (MHz)	132.2	131.6	115	130.6	101.8	131.2	88	82.5	80.2
Clock Period (ns)	7.56	7.60	8.70	7.66	9.82	7.62	11.36	12.12	12.47
#CC for unified multiplication	7	881	20401	33281	80801	181201	321601	502001	722401
Unifed multiplication time (ms)	0.00	0.01	0.18	0.25	0.79	1.38	3.65	6.08	9.01

polynomial multiplication by the unified multiplier. However, a combinational reduction circuit needs to be built into the core to reduce the partial products modulo p to prevent overflows.

The decryption of NTRU may be enhanced by choosing $f = 1 + p \cdot f_1$. In this case, the convolution $d = F_p * a \pmod p$ disappears and the first step becomes $a = f * e = e + p \cdot f_1 * e$. The convolution $f_1 * e$ is handled by our multiplier core; however, multiplication by p and the addition of e needs to be handled externally.

6 Performance Analysis

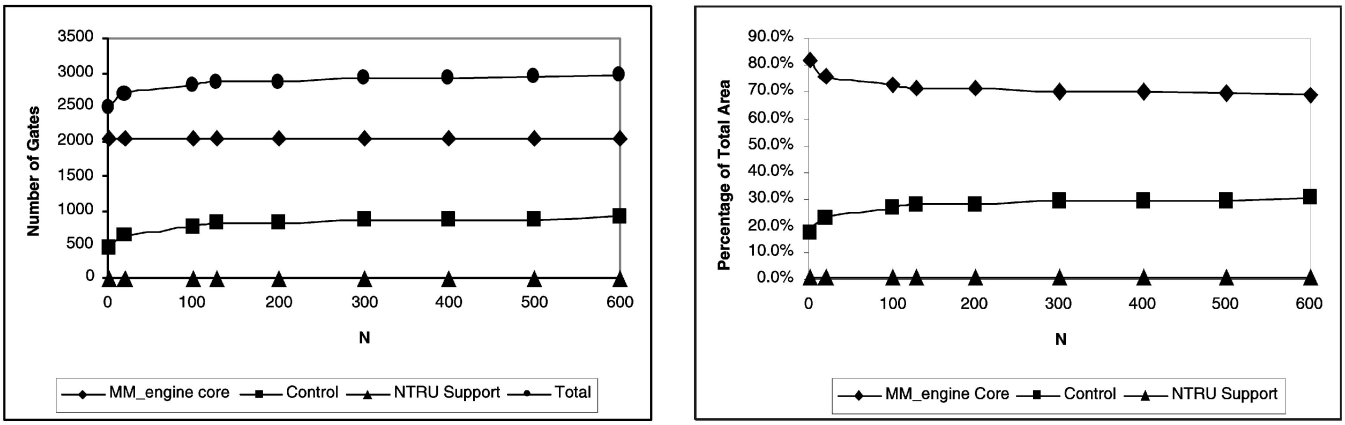
This section analyzes the performance of the unified multiplier. The architecture was modeled using VHDL, simulated for functionality using Mentor Graphics' ModelSim 5.5f, and synthesized with Mentor Graphics' LeonardoSpectrum tool using TSMC $0.35\mu m$ technology [11]. The data in Table 2 summarizes the overall numerical results for various performance criteria.

As seen in Fig. 4a, the total area increases slowly with the modulus length. This is due to the increase of the counters' size in the control unit. Despite this increase, the area scales at a slow rate and the majority of the area is used for the core, as seen in Fig. 4b. Due to the increase in control logic, the maximum clock frequency of the multiplier slightly decreases as the size of the modulus grows. Each stage of the control, as described in Section 4.3, is completed within a clock cycle. The total number of clock cycles (#CC) for an operand multiplication is determined by counting the number of stages executed. In Fig. 3, after a setup stage, both the inner loop and the outer loop are executed N times, which are two and six stages long, respectively. Hence, the total number clock cycles for a unified multiplication is

$$\#CC = (2N + 4)N + 1.$$

By multiplying #CC with the respective clock period, we obtain the timing results as shown in Table 2. For instance, a single unified multiplication for $N = 1$ is completed in 50 ns and for $N = 600$ in a little over 9 ms.

In Table 3, the performance of the unified multiplier is estimated for three cryptosystems, which represent each function of the multiplier. For the integer case, the unified multiplier can compute a 1,024-bit RSA operation with a short exponent in about 4 ms with less than $2,900$ gates, whereas the 1,024-bit RSA operation with a long exponent takes approximately 382 ms to complete with the same area. For the binary polynomial case, the unified multiplier requires approximately 15 ms to complete a 160-bit Elliptic Curve operation. It is assumed that projective coordinates



(a) (b)

Fig. 4. (a) Gate count and (b) percentage of total area for several operand lengths.

TABLE 3
Estimated Performance of Unified Design

1024-bit RSA	
short exponent	
# modular multiplications:	17
Time (ms):	4.33
long exponent	
# modular multiplications:	1500
Time (ms):	382.25
160-bit ECC	
# point doublings (9 mod mult per op):	159
# point additions (16 mod mult per op):	53
Total # modular multiplications:	2279
Time (ms):	15.26
503 NTRU	
Time (ms):	6.08

are used and the final inverse computation time is negligible. Finally, for NTRU's highest security level ($N = 503$), the unified multiplier can compute a polynomial multiplication in a little over 6 ms.

7 Future Improvements

This section outlines the potential improvements to the unified design. Specifically, the core can be further optimized for the NTRU mode. In this mode, $m[0] = -1$ and $m[0]' = 1$, therefore, Steps 3-5 in Algorithm 3.2 simplify as follows:

- Step 3: $U = S \cdot m[0]' \bmod 2^w = S$,
- Step 4: $CS = CS + (m[0] \cdot U) = CS - S = C$,
- Step 5: $CS = CS >> w = 0$.

Hence, the second multiplier can be bypassed. Ultimately, Step 3 becomes a simple assignment operation and Steps 4 and 5 can be merged to clear CS.

- Step 3: $U = S$,
- Step 4: $CS = 0$.

Furthermore, since NTRU's modulus is $x^N - 1$, within the inner loop, $m[i] = 0$ except in the last iteration, where $m[i] = 1$. This simplifies the $(m[i] \cdot U)$ portion of Step 7 to a single addition which only occurs during the last iteration of the inner loop. Therefore, the second multiplier can be bypassed in Step 7 as well since only an addition is necessary. Alternatively, the second multiplier may be utilized to process the next coefficient multiplication since NTRU's partial product columns are independent. Hence, the inner loop may execute twice as fast with additional control logic.

For proof of concept, the design was presented in its most simplified form with only one unified core. With less than $3,000$ gates, the footprint is significantly smaller than other unified Montgomery architectures. However, the performance is not comparable to such designs. If higher speeds are desired, the design may be scaled and pipelined at the expense of larger footprint. This would translate into a speedup increasing (almost) linearly with the number of cores.

Finally, the user may add a modulo p reduction circuit within the unified multiplier core to fully support NTRU's decryption procedure. For typical NTRU parameters $p = 3$ and $q = 256$, we were able to build a reduction unit at a cost of 46 two input gates (less than 2 percent of the overall design).

8 Conclusions

A unified architecture was designed to be simple and effective in demonstrating that NTRU's polynomial multiplication could easily be achieved with high-radix Montgomery Multipliers. The mathematical basis and algorithmic changes required for achieving NTRU with the MM algorithm were established. These modifications were implemented to produce a new unified architecture by extending a generic Montgomery Multiplier core with only nine additional gates. The unified multiplier provides support for all the polynomial multiplications required by NTRU's key creation and encryption procedures, as well as the decryption procedure with the addition of a modulo p reduction circuit. However, our design does restrict the integer modulus q to a power of 2.

The performance and application analysis conducted on the new unified core has demonstrated that all three types of applications, RSA, ECC, and NTRU, can be achieved with high performance on small footprint. It is interesting to note that 503 NTRU offers the best performance for its security level compared to the other two applications. For instance, NTRU with $N = 503$ provides a security level comparable to 4,096-bit RSA with an additional 2 ms over a short exponent 1,024-bit RSA operation. Despite the performance difference among these applications, the unified design is capable of supporting a majority of public-key cryptosystems such as NTRU [4], RSA [1], Diffie-Hellman [12], Elliptic Curves [2], [3], etc.

Acknowledgments

The authors thank Gunnar Gaubatz for his contributions to the unified Montgomery Multiplier core design. This material is based upon work supported by the US National Science Foundation under Grant No. ANI-0112889.

References

[1] R.L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Comm. ACM*, vol. 21, pp. 120-126, Feb. 1978.
[2] N. Koblitz, "Elliptic Curve Cryptosystems," *Math. Computation*, vol. 48, pp. 203-209, 1987.
[3] A.J. Menezes, *Elliptic Curve Public Key Cryptosystems*. Boston: Kluwer Academic, 1993.
[4] J. Hoffstein, J. Pipher, and J.H. Silverman, "NTRU: A Ring Based Public Key Cryptosystem," *Proc. Algorithmic Number Theory: Third Int'l Symp. (ANTS 3)*, J.P. Buhler, ed., pp. 267-288, June 1998.
[5] P.L. Montgomery, "Modular Multiplication without Trial Division," *Math. Computation*, vol. 44, pp. 519-521, Apr. 1985.
[6] Ç.K. Koç and T. Acar, "Montgomery Multplication in $GF(2^k)$," *Design, Codes, and Cryptography*, vol. 14, no. 1, pp. 57-69, 1998.
[7] E. Savas, A.F. Tenca, and Ç.K. Koç, "A Scalable and Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^m)$," *Proc. Workshop Cryptographic Hardware and Embedded Systems (CHES 2000)*, Ç.K. Koç and C. Paar, eds., pp. 277-292, 2000.

[8] G. Gaubatz, "Versatile Montgomery Multiplier Architectures," master's thesis, Electrical and Computer Eng. Dept., Worcester Polytechnic Inst., Worcester, Mass., Apr. 2002.
[9] J. Hoffstein and J.H. Silverman, "Optimizations for NTRU," *Proc. Public Key Cryptography and Computational Number Theory*, Sept. 2000.
[10] Ç.K. Koç, T. Acar, and B. Kaliski, "Analyzing and Comparing Montgomery Multiplication Algorithms," *IEEE Micro*, pp. 26-33, June 1996.
[11] M. Graphics, "ADK HTML Data Book TSMC 0.35 Micron FAST," 2001.
[12] W. Diffie and M.E. Hellman, "New Directions in Cryptography," *IEEE Trans. Information Theory*, vol. 22, pp. 644-654. 1976.

Colleen O'Rourke received the BSc degree in biomedical and electrical engineering in 2000 and the MS degree in electrical and computer engineering in 2002 from Worcester Polytechnic Institute (WPI). Her research interests include efficient arithmetic for lattice-based cryptosystems. The work presented in this paper was part of her master's thesis while at WPI. She is currently employed as an electrical engineer in Massachusetts at the Raytheon Company.

Berk Sunar received the BSc degree in electrical and electronics engineering from Middle East Technical University in 1995 and the PhD degree in electrical and computer engineering (ECE) from Oregon State University in December 1998. After briefly working as a member of the research faculty at Oregon State University, he joined Worcester Polytechnic Institute as an assistant professor. He currently heads the Cryptography and Information Security Laboratory (CRIS). His research interests include finite fields, elliptic curve cryptography, computer arithmetic, and energy efficient crypographic hardware. He is a member of the IEEE, the IEEE Computer Society, the ACM, and the International Association of Cryptologic Research (IACR).