Efficient NTRU Implementations

A Thesis
submitted to the Faculty
of the
Worcester Polytechnic Institute
In partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical & Computer Engineering
by

Colleen Marie O'Rourke

April 2002

Approved:

Dr. Berk Sunar Thesis Advisor ECE Department	Dr. Donald Brown Thesis Committee ECE Department
Dr. Fred Looft Thesis Committee ECE Department	Dr. John Orr Department Head ECE Department

Abstract

In this paper, new software and hardware designs for the NTRU Public Key Cryptosystem are proposed. The first design attempts to improve NTRU's polynomial multiplication through applying techniques from the Chinese Remainder Theorem (CRT) to the convolution algorithm. Although the application of CRT shows promise for the creation of the inverse polynomials in the setup procedure, it does not provide any benefits to the procedures that are critical to the performance of NTRU (public key creation, encryption, and decryption). This research has identified that this is due to the small coefficients of one of the operands, which can be a common misunderstanding.

The second design focuses on improving the performance of the polynomial multiplications within NTRU's key creation, encryption, and decryption procedures through hardware. This design exploits the inherent parallelism within a polynomial multiplication to make scalability possible. The advantage scalability provides is that it allows the user to customize the design for low and high power applications. In addition, the support for arbitrary precision allows the user to meet the desired security

level.

The third design utilizes the Montgomery Multiplication algorithm to develop an unified architecture that can perform a modular multiplication for $GF(p)$ and $GF(2^k)$ and a polynomial multiplication for NTRU. The unified design only requires an additional 10 gates in order for the Montgomery Multiplier core to compute the polynomial multiplication for NTRU. However, this added support for NTRU presents some restrictions on the supported lengths of the moduli and on the chosen value for the residue for the $GF(p)$ and $GF(2^k)$ cases. Despite these restrictions, this unified architecture is now capable of supporting public key operations for the majority of Public-Key Cryptosystems.

Preface

In this work, I detail the research I have conducted in pursuit of my Master's Degree at Worcester Polytechnic Institute.

This work would not have possible without the guidance and support of my advisor, Prof. Berk Sunar. He gave me the opportunity to explore and perform research in field of cryptography. Working with Prof. Sunar has been such a learning experience and I hope to keep in touch.

I would like to thank my Thesis committee, Profs. Donald R. Brown and Fred Looft, for their advice and support. They showed enthusiasm about my research and being a part of my committee.

In addition, I would like to thank the Cryptography Lab, Gunnar Gaubatz, Adam Elbirt and Selcuk Baktir, for providing a great working environment and assisting me with any questions I may have had. I am especially grateful to Gunnar Gaubatz who gave me his VHDL design of his Montgomery Multiplier core. As a result of his contribution, I was able to take my research one step further and expand off his design to develop my Unified NTRU & Montgomery Multiplier.

Finally, I would like to dedicate this thesis to my father, James O'Rourke, and my boyfriend of four years, Jeffrey Jacobson. They have provided me with unwavering support and encouragement during all my years here at Worcester Polytechnic Institute. I want to thank them both for always being there for me.

Colleen M. O'Rourke

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The NTRU Public-Key Cryptosystem is a ring-based cryptosystem that was first introduced in a rump session at Crypto' 96 and the first formal paper was published in CHES '98 [3]. NTRU is a relatively new cryptosystem that appears to be more efficient than the current and more widely used public-key cryptosystems, such as RSA [4]. In comparison to RSA, NTRU requires approximately $\mathcal{O}(N^2)$ operations and a key length of $\mathcal{O}(N)$, whereas RSA requires $\mathcal{O}(N^3)$ operations and a key length of $\mathcal{O}(N^2)$ [3]. Hence, NTRU has a lower complexity and its key size scales at a slower rate. Although NTRU's key size may be longer, it can be implemented more efficiently than RSA because it has a smaller number of multiplications for encryption and decryption. Therefore, this cryptosystem is a promising alternative to the more established public-key cryptosystems. However, these more established cryptosystems have received more public scruntity, whereas, NTRU is still a relatively young cryptosystem.

NTRU's core and most time consuming operation for key creation, encryption, and decryption is the multiplication of two polynomials of degree $N-1$ defined over an integer ring. In order to improve the performance of NTRU, it is necessary to develop software algorithms or a hardware accelerator that will speed up the polynomial multiplications. In the past couple of years, a few software improvements have been published that improve the convolution algorithm used for polynomial multiplication. However, to date, there has been only one published research [2] that implemented NTRU in hardware. This research focused solely on an encryption engine implemented on an FPGA. The design is not scalable and cannot be used for key creation or decryption. Since no research has been published that soley enhances the performance of the polynomial multiplication in hardware, this presents an opportunity for this research.

With the knowledge gained from designing an optimized, scalable NTRU polynomial multiplier, this thesis took the next step to develop an unified architecture that performs modular multiplications in $GF(p)$ and $GF(2^k)$ as well as NTRU's polynomial multiplications. The importance of an unified design is that it allows for:

- algorithm agility,
- resource utilization, and
- compatibility.

Algorithm agility is useful for cases when the current cryptosystem becomes obsolete

or when computing power increases and the security of the current cryptosystem is threatened. With algorithm agility, there will be several back-up cryptosystems, which utilizes the same hardware, that can be used as a replacement if the case ever arises. Compatibility is another important feature since it allows support for virtually any application. For instance, wireless applications may want to utilize cryptosystems such as, ECC [5, 6], whereas, for smart cards or for other embedded applications NTRU may be the more reasonable choice. Without the hassle of designing a separate chip for each application, this unified design supports a wide variety of applications on a single chip.

A number of unified architectures have been proposed that use the Montgomery Multiplication algorithm for modular multiplications in $GF(p)$ and $GF(2^k)$. However, this is the first realization to incorporate NTRU's polynomial multiplication using the Montgomery Multiplication algorithm.

This report is divided into three parts so that the software and each of the hardware designs can be discussed separately. Each part contains previous work, background information, design details, and a performance analysis.

Chapter 2

NTRU Public Key Crytosystem

The NTRU Public Key Cryptosystem was fully introduced in [3]. NTRU 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 the "$*$" 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. More

information about NTRU can be found in [3] and [7]. The procedures are briefly outlined below.

2.1 Key Generation

For the public key, the user must:

- choose a secret key, a random secret polynomial $f \in R$, with coefficients reduced modulo p,
- choose a random polynomial, $g \in R$, with coefficients reduced modulo p, 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

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

2.2 Encryption

The encrypted message is computed as

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

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

2.3 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 ($d = m$).

Part I

Software Implementations

Chapter 3

Previous Work

This section introduces NTRU's polynomial multiplication as presented in [3] and discusses the two previous contributions that focuses on enhancing the performance of NTRU's polynomial multiplication through software. When NTRU was formally introduced in 1998 [3], Silverman presented the polynomial multiplication as the cyclic convolution of two polynomials as shown below:

$$c_k = \sum_{i=0}^{k} a_i \cdot b_{k-i} + \sum_{i=k+1}^{N-1} a_i \cdot b_{N+k-i} = \sum_{i+j=k \pmod N} a_i \cdot b_j$$

In addition, Silverman presented this convolution algorithm in [8], which is shown in Algorithm 3. Ultimately, this straightforward method requires N^2 multiplications to perform a polynomial multiplication for NTRU.

In 1999, Silverman applies a technique presented in [9] to improve NTRU's polynomial multiplication [1]. Basically, the idea involves sucessively splitting the two polynomial operands in half via the recursive algorithm, which is based off the Karatsuba-

Ofman algorithm [10], shown in Algorithm 1. The polynomials are continually split in half by Steps 8-19 of Algorithm 1 until the degree of the divided polynomials is less than *CutOff*, which is defined by the user to comply with any design constraints. At this point, the product is computed as a convolution by Steps 1-7. If the degree of c is larger than N, then Steps 20-24 wraps the terms exceeding a degree of x^{N-1} to the lower portion by using the property $x^N \equiv 1 \pmod{x^N - 1}$. If Algorithm 1 is recursively called r times, then the number of operations to perform NTRU's polynomial multiplication is effectively reduced to $(\frac{3}{4})^r N^2$. This presents great savings over the straightforward method.

Algorithm 1 PolyMult(c, b, a, n, N) from [1]

Require: N, n, and the polynomial operands, a and b.
1: **if** $n < CutOff$ **then**
2: **for** $k = 0$ to $2 \cdot n - 2$ **do**
3: $c[k] = 0$
4: **for** $i = max(0, k - n + 1)$ to $min(k, n - 1)$ **do**
5: $c[k] = c[k] + b[i] \cdot a[k - i]$
6: **end for**
7: **end for**
8: **else**
9: $n1 = n/2$
10: $n2 = n - n1$
11: $b = b1 + b2 \cdot x^{n1}$
12: $a = a1 + a2 \cdot x^{n1}$
13: $B = b1 + b2$
14: $A = a1 + a2$
15: **PolyMult**$(c1, b1, a1, n1, N)$
16: **PolyMult**$(c2, b2, a2, n2, N)$
17: **PolyMult**$(c3, B, A, n2, N)$
18: $c = c1 + (c3 - c1 - c2) \cdot x^{n1} + c2 \cdot x^{2 \cdot n1}$
19: **end if**
20: **if** $2 \cdot n - 1 > N$ **and** $N > 0$ **then**
21: **for** $k = N$ to $2 \cdot n - 2$ **do**
22: $c[k - N] = c[k - N] + c[k]$
23: **end for**
24: **end if**
25: {PolyMult returns the product polynomial, c, throught the argument list.}

Finally, in 2001, Bailey et al. [2] introduced a fast convolution algorithm, which is shown in Algorithm 2, to perform a polynomial multiplication for NTRU. This algorithm makes the realization that almost every polynomial multiplication involved with NTRU has one polynomial that is random. The random polynomials are assumed to have binary coefficients. In addition, the random polynomial is assumed to consist of three smaller polynomials of low Hamming weight [11]:

$$f(x) = f_1(x) * f_2(x) + f_3(x)$$

The number of 1's in each of the smaller polynomials is represented as d_1, d_2, and d_3, respectively. This fast convolution algorithm reduces the complexity of NTRU's polynomial multiplication to $(d_1 + d_2 + d_3)N$ additions and no multiplications without compromising the security.

Algorithm 2 Fast Convolution Algorithm from [2]

Require: b an array of $d_1+d_2+d_3$ nonzero coefficient locations representing the polynomial $f(x) = f_1(x) * f_2(x) + f_3(x)$, a the array $a(x) = \sum a_i$, N the number of coefficients in $f(x)$ and $a(x)$, q the modulus of the integer operations.

Ensure: c the array where $c(x) = f(x) * a(x)$

1: {Compute $t(x) = a(x) * f_1(x)$}
2: **for** $j = 0$ to $d_1 - 1$ **do**
3: **for** $k = 0$ to $N - 1$ **do**
4: $t[k + b[j]] = t[k + b[j]] + a[k] \bmod q$
5: **end for**
6: **end for**
7: {Compute $c(x) = t(x) * f_2(x) = a(x) * f_1(x) * f_2(x)$}
8: **for** $j = d_1$ to $d_2 - 1$ **do**
9: **for** $k = 0$ to $N - 1$ **do**
10: $c[k + b[j]] = c[k + b[j]] + t[k] \bmod q$
11: **end for**
12: **end for**
13: {Zero out $t(x)$}
14: **for** $k = 0$ to $N - 1$ **do**
15: $t[k] = 0$
16: **end for**
17: {Compute $t(x) = a(x) * f_3(x)$}
18: **for** $j = d_2$ to $d_3 - 1$ **do**
19: **for** $k = 0$ to $N - 1$ **do**
20: $t[k + b[j]] = t[k + b[j]] + a[k] \bmod q$
21: **end for**
22: **end for**
23: {Compute $c(x) = c(x) + t(x)$}
24: **for** $k = 0$ to $N - 1$ **do**
25: $c[k] = c[k] + t[k] \bmod q$
26: **end for**

Chapter 4

Initial Implementation

In order to gain a full understanding of how the NTRU Cryptosystem functioned, this cryptosystem is first implemented in software using the C Language. For this implementation, the code was designed for $p = 3$ since this was the value introduced in [3]. As an interface, the user was allowed to specify the integer parameters N, q, and the number of ones and negative ones that would make up the random polynomials f, g, and r, respectively. The pseudo-code presented in [8] was used as a guideline to develop the **CreateKey**, **Encode**, **Decode**, and **StarMultiply** functions. Also, the techniques introduced in [12] were utilized to design functions that would create inverse polynomials for the secret key modulo q and modulo p. Lastly, a **RandPoly** function was written to generate the random polynomials that were necessary within NTRU's procedures. The details and the purpose of these functions are explained in more detail below.

StarMultiply:

This function outlined in Algorithm 3 performs the polynomial multiplication of $a * b \bmod x^N - 1$. As a note, the M in Step 9 is either p or q depending upon which one is passed into the function. In contrast to the guideline in [8], Algorithm 3 only executes Step 9 if the current coefficients of $a[i]$ and $b[j]$ are both non-zero. This, therefore, eliminates approximately a third of the operations, which are unnecessary. Also, for the case $M = q$, Algorithm 3 assumes $q = 2^w$ so the reduction is performed by extracting the lower w bits.

RandPoly:

The **RandPoly** function, shown in Algorithm 4, generates a random polynomial, r, whose coefficients are in the subset {-1,0,1}. The user specifies the number of ones ($NumOnes$) and the number of negative ones ($NumNegOnes$) that will make up the random polynomial, r. Basically, the algorithm works by randomly selecting a location (*position*) between 0 and N in the random polynomial vector, r. For each selected location, if the value is zero the algorithm replaces the zero with a 1 or a -1 until all the specified number of ones and negative ones have been entered into the vector.

Inverse_Poly_Fq:

The **Inverse_Poly_Fq** function in Algorithm 5 is responsible for generating the inverse polynomial of the secret key, f, modulo q. The first 40 lines of Algorithm 5 computes the inverse of the secret key modulo 2. Then, the last couple of lines in the

Algorithm 3 StarMultiply(a, b, c, N, M)

Require: N, the coefficient modulus, M, and the two polynomials to be multiplied, a and b.

```
1: for k = N − 1 downto 0 do
2:    c[k] = 0
3:    j = k + 1
4:    for i = N − 1 downto 0 do
5:       if j = N then
6:          j = 0
7:       end if
8:       if a[i] ≠ 0 and b[j] ≠ 0 then
9:          c[k] = c[k] + (a[i] · b[j]) mod M
10:      end if
11:      j = j + 1
12:   end for
13: end for
14: {StarMultiply returns the product polynomial, c, throught the argument list.}
```

Algorithm 4 RandPoly(r, N, *NumOnes*, *NumNegOnes*)

Require: N, *NumOnes*, *NumNegOnes*, and polynomial vector to be made random, r.

```
1: r = 0
2: while NumOnes ≠ 0 or NumNegOnes ≠ 0 do
3:    position = rand() mod N
4:    if r[position] = 0 then
5:       if NumOnes > 0 then
6:          r[position] = 1
7:          NumOnes = NumOnes − 1
8:       else if NumNegOnes > 0 then
9:          r[position] = −1
10:         NumNegOnes = NumNegOnes − 1
11:      end if
12:   end if
13: end while
14: {RandPoly returns the newly generated random polynomial, r, through the argument list}
```

algorithm finds the inverse polynomial modulo a power of 2, which is q. Algorithm 5 is based off the pseudo-code for "Inversion in $(Z/2Z)[X]/(X^N - 1)$" and "Inversion in $(Z/p^rZ)[X]/(X^N - 1)$" provided in [12]. Please seek this reference for more detail on how this algorithm functions.

Inverse_Poly_Fp:

The **Inverse_Poly_Fp** function in Algorithm 6 is responsible for generating the inverse polynomial of the secret key, f, modulo p. Algorithm 6 is based off the pseudo-code for "Inversion in $(Z/pZ)[X]/(X^N - 1)$" provided in [12]. Please seek this reference for more detail on how this algorithm functions.

CreateKey:

The **CreateKey** function, shown in Algorithm 7, is responsible for:

1. creating the inverse polynomial of the secret key modulo q, F_q (Step 3),
2. creating the inverse polynomial of the secret key modulo p, F_p (Step 4), and
3. creating the Public Key, $h = p \cdot (F_q * g) \bmod q$ (Steps 8-14).

Also, Algorithm 7 assumes $q = 2^w$ so the reduction in Step 13 is performed by extracting the lower w bits.

Algorithm 5 Inverse_Poly_Fq(a, F_q, N, q)

Require: the polynomial to invert $a(x)$, N, and q.

1: $k = 0$
2: $b = 1$
3: $c = 0$
4: $f = a$
5: $g = 0$ {Steps 5-7 set g(x) = $x^N - 1$.}
6: $g[0] = -1$
7: $g[N] = 1$
8: **loop**
9: **while** $f[0] = 0$ **do**
10: **for** $i = 1$ to N **do**
11: $f[i-1] = f[i]$ {f(x) = f(x)/x}
12: $c[N+1-i] = c[N-i]$ {c(x) = c(x) · x}
13: **end for**
14: $f[N] = 0$
15: $c[0] = 0$
16: $k = k + 1$
17: **end while**
18: **if** deg(f) = 0 **then**
19: **goto** Step 32
20: **end if**
21: **if** $deg(f) < deg(g)$ **then**
22: $temp = f$ {Exchange f and g}
23: $f = g$
24: $g = temp$
25: $temp = b$ {Exchange b and c}
26: $b = c$
27: $c = temp$
28: **end if**
29: $f = f \oplus g$
30: $b = b \oplus c$
31: **end loop**
32: $j = 0$
33: $k = k \bmod N$
34: **for** $i = N - 1$ downto 0 **do**
35: $j = i - k$
36: **if** $j < 0$ **then**
37: $j = j + N$
38: **end if**
39: $F_q[j] = b[i]$
40: **end for**
41: $v = 2$
42: **while** $v < q$ **do**
43: $v = v * 2$
44: **StarMultiply**$(a, F_q, temp, N, v)$
45: $temp = 2 - temp \bmod v$
46: **StarMultiply**$(F_q, temp, F_q, N, v)$
47: **end while**
48: **for** $i = N - 1$ downto 0 **do**
49: **if** $F_q[i] < 0$ **then**
50: $F_q[i] = F_q[i] + q$
51: **end if**
52: **end for**
53: {Inverse_Poly_Fq returns the inverse polynomial, F_q, through the argument list.}

Algorithm 6 Inverse_Poly_Fp(a, F_p, N, p)

Require: the polynomial to invert $a(x)$, N, and p.

1: $k = 0$
2: $b = 1$
3: $c = 0$
4: $f = a$
5: $g = 0$ {Steps 5-7 set g(x) = $x^N - 1$.}
6: $g[0] = -1$
7: $g[N] = 1$
8: **loop**
9: **while** $f[0] = 0$ **do**
10: **for** $i = 1$ to N **do**
11: $f[i-1] = f[i]$ {f(x) = f(x)/x}
12: $c[N+1-i] = c[N-i]$ {c(x) = c(x) · x}
13: **end for**
14: $f[N] = 0$
15: $c[0] = 0$
16: $k = k + 1$
17: **end while**
18: **if** deg(f) = 0 **then**
19: **goto** Step 33
20: **end if**
21: **if** $deg(f) < deg(g)$ **then**
22: $temp = f$ {Exchange f and g}
23: $f = g$
24: $g = temp$
25: $temp = b$ {Exchange b and c}
26: $b = c$
27: $c = temp$
28: **end if**
29: $u = f[0] \cdot g[0]^{-1} \bmod p$
30: $f = f - u \cdot g \bmod p$
31: $b = b - u \cdot c \bmod p$
32: **end loop**
33: $j = 0$
34: $k = k \bmod N$
35: **for** $i = N - 1$ downto 0 **do**
36: $b[i] = f[0]^{-1} \cdot b[i] \bmod p$
37: $j = i - k$
38: **if** $j < 0$ **then**
39: $j = j + N$
40: **end if**
41: $F_q[j] = b[i]$
42: **end for**
43: {Inverse_Poly_Fp returns the inverse polynomial, F_p, through the argument list.}

Algorithm 7 CreateKey($N, q, p, f, g, h, F_p, F_q$)

Require: p, q, N and random polynomials, f and g.
1: **Inverse_Poly_Fq**(f, F_q, N, q)
2: **Inverse_Poly_Fp**(f, F_p, N, p)
3: **StarMultiply**(Fq, g, h, N, q)
4: **for** $i = 0$ to $N - 1$ **do**
5: **if** $h[i] < 0$ **then**
6: $h[i] = h[i] + q$ {Make all coefficients in h positive.}
7: **end if**
8: $h[i] = h[i] \cdot p \bmod q$
9: **end for**
10: {CreateKey returns the Public Key, h, and the inverse polynomial, F_p, through the argument list.}

Encode:

The **Encode** function in Algorithm 8 creates the encrypted message, $e = (h * r) + m \bmod q$. This is accomplished by:

1. performing the polynomial multiplication of $h * r$ in Step 1 and
2. adding the message m in Steps 2-4.

Again, the modulo reduction in Step 3 is performed by extracting the lower w bits.

Algorithm 8 Encode(N, q, r, m, h, e)

Require: N, q, Public Key h, message m, and random polynomial r.
1: **StarMultiply**(r, h, e, N, q)
2: **for** $i = 0$ to $N - 1$ **do**
3: $e[i] = e[i] + m[i] \bmod q$
4: **end for**
5: {Encode returns the encrypted message, e, through the argument list.}

Decode:

The **Decode** function in Algorithm 9 is responsible for decrypting the encrypted message. The decryption procedure is executed by the following three steps:

1. performing the polynomial multiplication of $a = f * e \bmod q$ (Step 1),

2. shifting the coefficients of a into the range $(-q/2, q/2)$ (Steps 2-9), and

3. performing the polynomial multiplication of $d = a * F_p \bmod p$ (Step 10).

Algorithm 9 Decode(N, q, p, f, F_p, e, d)

Require: N, q, p, secret key f, inverse polynomial F_p, and encrypted message e.
1: **StarMultiply**(f, e, a, N, q)
2: **for** $i = 0$ to $N - 1$ **do**
3: **if** $a[i] < 0$ **then**
4: $a[i] = a[i] + q$ {Make all coefficients positive}
5: **end if**
6: **if** $a[i] > q/2$ **then**
7: $a[i] = a[i] - q$ {Shift coefficients of a into range $(-q/2, q/2)$}
8: **end if**
9: **end for**
10: **StarMultiply**(a, F_p, d, N, p)
11: {Decode returns the decrypted message, d, through the argument list.}

4.1 Performance Analysis

In order to grasp how well NTRU performs for different applications, a timing analysis was conducted for the **CreateKey**, **Encode**, and **Decode** functions on two different platforms outlined in Tables 4.1. In addition, this timing analysis considered two levels of security for NTRU, which were the lowest and highest security recommended in [3]. The test values for the parameters of NTRU used for this performance analysis are listed in Table 4.2.

	First Platform	**Second Platform**
Processor	266 MHz PentiumII	50 MHz ARM7TDMI
Memory	65 MB	—
OS	Windows 98SE	—
Compiler	MS Visual C++	ARM Developmental Suite

Table 4.1: Two Platform Specifications

	107 NTRU	**503 NTRU**
N	107	503
q	64	256
p	3	3
NumOnes f	15	216
NumNegOnes f	14	215
NumOnes g	12	72
NumNegOnes g	12	72
NumOnes r	5	55
NumNegOnes r	5	55
NumOnes m	25	165
NumNegOnes m	25	165

Table 4.2: Test Values Used for Performance Analysis

The results of the timing analysis are shown in Table 4.3.

	266 MHz PII		50 MHz ARM7TDMI	
	107 NTRU	503 NTRU	107 NTRU	503 NTRU
CreateKey (ms)	16.2	699.5	91.4	2412.1
Encode (ms)	0.6	15.0	4.9	110.9
Decode (ms)	1.4	29.4	5.7	163.1

Table 4.3: Timings for NTRU on Two Platforms

The **CreateKey** function takes the longest time because it requires two polynomial inversions and a polynomial multiplication. In addition, since the **Decode** function requires two polynomial multiplications, it takes over two times as long as the **Encode** function. Since **Encode** requires only one polynomial multiplication, it is fair to use **Encode**'s timing to estimate the time to perform a single polynomial multiplication. Altogether, the timing analysis in Table 4.3 shows that NTRU has potential in offering high performance.

Chapter 5

Application of Chinese Remainder Theorem

Similar to the efforts of Bailey et al. and Silverman in Chapter 3, this thesis aimed to improve NTRU's polynomial multiplication through software by improving the convolution algorithm. So, for the specific case $p = 3$, the research explored the possibility of improving the convolution by applying techniques from the Chinese Remainder Theorem (CRT).

5.1 Chinese Remainder Theorem

CRT [13] can simplify the operands and modulus used for the polynomial multiplication by factorizing the modulus, $M = x^N - 1$. If M has t factors such that:

- $M = \prod_{i=0}^{t-1} m_i$ and

- $\gcd(m_i, m_j) = 1$ for all $i \neq j$,

then an operand, a, can be broken down into smaller polynomials as well:

$$\begin{aligned} a &\equiv a_0 \pmod{m_0} \\ a &\equiv a_1 \pmod{m_1} \\ &\vdots \\ a &\equiv a_{t-1} \pmod{m_{t-1}} \end{aligned}$$

As a result of CRT, the convolution can now be performed on each of the smaller polynomials as follows:

$$\begin{aligned} c_0 &= a_0 * b_0 \pmod{m_0} \\ c_1 &= a_1 * b_1 \pmod{m_1} \\ &\vdots \\ c_{t-1} &= a_{t-1} * b_{t-1} \pmod{m_{t-1}} \end{aligned}$$

This therefore reduces the number of multiplications to the sum of the squares of the degrees of the factors of M $(N_0^2 + N_1^2 + \cdots + N_{t-1}^2)$. The computational cost of the sum of smaller squares, which are smaller than the degree of M (N), is less than the straightforward convolution of two large operands N^2. As a note, the reductions in converting the operands to the CRT residue system can be computed as a series of additions and subtractions.

5.2 Useful with NTRU?

Ultimately, CRT replaces a small number of integer multiplications with a large number of additions. Although CRT looks promising for reducing the number of multiplications, the question is whether this technique will be useful for NTRU's polynomial

multiplication for the case $p = 3$. In NTRU's main operations (public key creation, encryption, and decryption), most of the polynomial multiplications consist of one operand that has coefficients in the subset {-1,0,1}. So, in reality, the convolution of two polynomials with one operand with this property is simply a series of additions and subtractions and no multiplications. Therefore, for the majority of NTRU's polynomial multiplicatons, applying CRT to the convolution provides no benefits. However, this CRT based convolution can be used in the setup phase for the creation of the inverse polynomials. As presented in Algorithms 5 and 6, the inverse polynomial functions rely on the polynomial multiplication of two polynomials with w-bit coefficients. One may argue that the CRT technique can be applied to the last step of decryption since it requires the multiplication of two polynomials one of which has w-bit coefficients and the other which has coefficients in the subset {0,1,2}. However, the convolution of these two polynomials can be performed as a series of additions, subtractions, or shift operations instead of multiplications because of the small coefficients within the second polynomial operand. So, once again, there are no multiplications that CRT can replace with a large number of additions. Since the application of the CRT based convolution to NTRU's polynomial multiplication does not provide any improvement to the core operations in the public key creation, encryption, and decryption procedures, this thesis decided not to pursue this avenue any further.

Part II

Scalable NTRU Multiplier

Chapter 6

Previous Work

This section reviews the only previous contribution to the design of a hardware implementation for NTRU [2]. Bailey et al. designed an encryption engine on a Xilinx Virtex 1000EFG860 FPGA [14] platform that performed the polynomial multiplication of $r * h$ and the addition of the message, m. The system required that all three operands be serially loaded in the FPGA. Then, a fast convolution algorithm is utilized to perform the polynomial multiplication of $r * h$. Further reading and details on this algorithm is discussed in [2] as well as Chapter 3. Finally, the encryption engine adds the result of the polynomial multiplication with the message, m, and serially outputs the encrypted message. The advantage of this design is that Bailey et al. enhanced the performance of NTRU's encryption procedure. Yet, the weaknesses of this design are that it is not scalable and it cannot be used for any other procedure (key creation and decryption) within NTRU.

Chapter 7

Hardware Design

This paper will present an optimized and scalable hardware design that can perform a polynomial multiplication for NTRU. We fix the parameters p and q to 3 and 256, respectively. However, this design supports arbitrary polynomial lengths including $N = 503$, which is considered to provide a security level comparable to 4096-bit RSA [2]. In addition, the design takes advantage of the parallel nature of the partial product array. Since all polynomials are reduced modulo $x^N - 1$, the partial product terms exceeding x^{N-1} will be wrapped around and added to the lower portion. Figure 7.1 shows the case $N = 5$, where the coefficients of the polynomial are represented as coefficient arrays (e.g. $a(x) = \sum_{i=0}^{N-1} a_i x^i \leftrightarrow a = (a_{N-1}, \cdots, a_1, a_0)$).

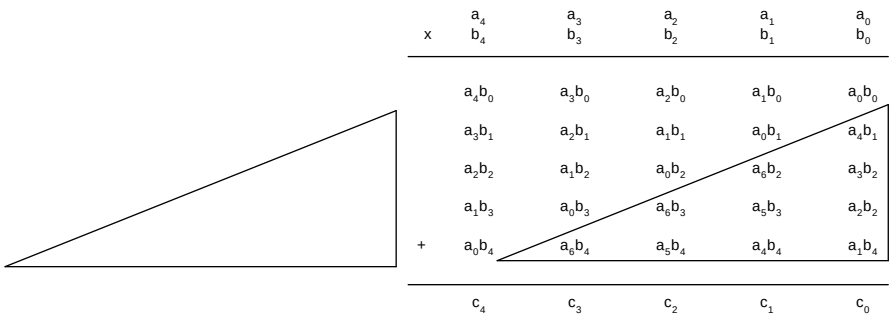


Figure 7.1: NTRU Multiplier

Since each partial product term is reduced modulo q, which is the lower word, then carry propagation is confined within each column but not across the columns.

7.1 Scalability

Due to the absence of carry propagation across the columns, the independent columns within the partial product array can be processed in parallel. For the partial product column k, a single processing unit (PU) performs the following operation:

$$\begin{aligned} c[k] = c[k] + a[i] \cdot b[j] \pmod{q} \quad j &= 0, 1, \ldots, N-1 \\ i &= -j \bmod N \end{aligned} \tag{7.1}$$

which consists of one coefficient multiplication, one coefficient addition, and a reduction modulo q. Parallelism can, therefore, be achieved by scaling the number of processing units (u) across the partial product columns, which is represented by the boxes in Figure 7.2. Since each processing unit works independently of the other,

multiple partial product columns can be processed concurrently.

	a_6	a_5	a_4	a_3	a_2	a_1	a_0	
$\times$	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
	a_6b_0	a_5b_0	a_4b_0	a_3b_0	$\boxed{a_2b_0}$	$\boxed{a_1b_0}$	$\boxed{a_0b_0}$	
	a_5b_1	a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1	a_6b_1	$\vert$
	a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2	a_6b_2	a_5b_2	$\vert$
	a_3b_3	a_2b_3	a_1b_3	a_0b_3	a_6b_3	a_5b_3	a_4b_3	$\vert$
	a_2b_4	a_1b_4	a_0b_4	a_6b_4	a_5b_4	a_4b_4	a_3b_4	$\vert$
	a_1b_5	a_0b_5	a_6b_5	a_5b_5	a_4b_5	a_3b_5	a_2b_5	$\vert$
$+$	a_0b_6	a_6b_6	a_5b_6	a_4b_6	a_3b_6	a_2b_6	a_1b_6	$\vee$
	c_6	c_5	c_4	c_3	c_2	c_1	c_0	

Figure 7.2: Partial product array for $N = 7$ and $u = 3$.

In this paper, we define scalable as the ability to replicate or reuse a processing unit *during design time* in order to perform multiple operations in parallel without re-designing the data path of the processing unit. We define arbitrary precision as the ability to alter the length of the operand polynomials *during run time* up to the maximum size supported by the counters within the control unit.

7.2 Scalable Architecture

The scalable NTRU multiplier shown in Figure 7.3 consists of three major components, the control unit, the registers, and a row of processing units (processing row). The control unit orchestrates the computation of the polynomial multiplication and interacts with memory via the host system. There are four major registers for the multiplier: two $u \times 8$-bit shift registers, a 2×2-bit shift register, and a $u \times 8$-bit register. Before implementation, the user configures the number of units that makes up the processing row. The design assumes that the coefficients for each polynomial

(a, b, and c) reside in separate memory caches, which are not part of this design. In addition, it is required that there be $u + (\lfloor \frac{N+u-1}{u} \rfloor \cdot u - N)$ additional slots in $Cache\ A$ and $\lfloor \frac{N+u-1}{u} \rfloor \cdot u - N$ additional slots in $Cache\ C$. We also assume that a memory read or write can be performed in less than one clock cycle.

The typical sequence of steps conducted by the control unit to perform a single polynomial multiplication is presented in Algorithm 10. Within the algorithm,

- u is the number of processing units,
- A is a vector that holds u 8-bit coefficients for the polynomial a,
- B is a vector that holds two 2-bit coefficients for the polynomial b,
- C is a vector that holds u 8-bit coefficients for the intermediate result, and
- $Save$ is a vector that holds u 8-bit coefficients for the final result.

The "$\cdot$" denotes that each element in vector A is multiplied with B_1 and reduced modulo q separately. In addition, the "+" denotes that each element of the vector C is added with its respective element from the result of $(A \cdot B_1 \bmod q)$.

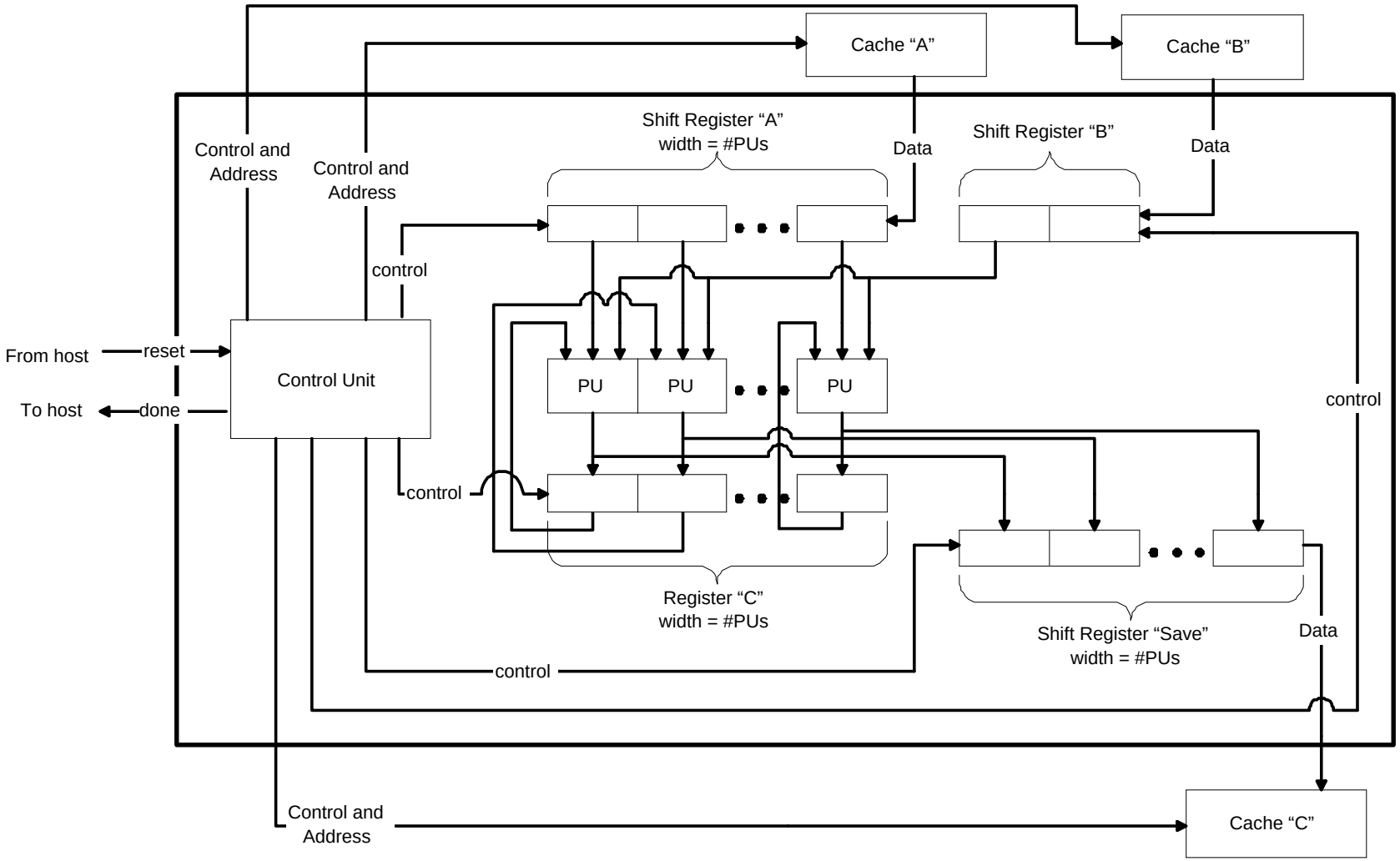


Figure 7.3: NTRU Multiplier

Algorithm 10 NTRU Polynomial Multiplication

1: **if** $reset = `1'$ **then**
2: $C = 0$
3: $Save = 0$
4: $x = 0$ {x is a counter}
5: **else**
6: $A = (a[u-1], \cdots, a[0])$
7: $B = (b[0], b[1])$
8: **for** $j = 1$ to $\lceil N/u \rceil$ **do**
9: **for** $i = 1$ to $N - 1$ **do**
10: $C = C + (A \cdot B_1) \bmod q$ {B_1 represents upper word of B}
11: $A = (A_{u-2} \cdots A_0, a[N-i-1])$
12: $B = (B_0, b[i+1 \bmod N])$
13: **end for**
14: $Save = C + (A \cdot B_1) \bmod q$
15: $C = 0$
16: $(c[x+u], \cdots, c[x]) = Save$
17: $x = x + u$
18: $A = (a[x-1], \cdots, a[x-u])$
19: $B = (B_0, b[1])$
20: **end for**
21: **end if**

To demonstrate how this algorithm applies to the hardware shown in Figure 7.3, we will go through the steps for the example shown in Figure 7.2. For clarity, the description below assumes that:

- the multiplier is configured for three processing units ($u = 3$),
- the operands, polynomials a and b, and the result, polynomial c, each have seven coefficients ($N = 7$),
- the coefficient arrays are now represented as $a(x) = \sum_{i=0}^{N-1} a[i]x^i \leftrightarrow a = (a[N-1], \cdots, a[1], a[0])$,
- the vectors A, B, C, and $Save$ in Algorithm 10 correspond to the respective registers in Figure 7.3, and

- the PUs will process $\lfloor \frac{N+u-1}{u} \rfloor$ sets of u columns in the partial product array.

In order to take advantage of concurrency, Algorithm 10 is unrolled to execute multiple steps in parallel within hardware and to split the algorithm into three stages, the initialization stage, the execution stage, and the load/write stage. The initialization stage executes steps 1 through 7 in Algorithm 10. Whereas, the processing stage executes steps 9 through 13. Finally, the load/write stage executes steps 14 through 19 of Algorithm 10.

After the user configures the degree N of the modulus, the *reset* signal is asserted by the host system to inform the NTRU multiplier to begin execution of a polynomial multiplication. At this point, the control unit begins the initialization stage, which is outlined in Table 7.1. First, at clock cycle 0 (CC0), registers C and $Save$ are cleared. At CC1, the host system sets the reset signal low and the control unit begins to serially load the first three (u) coefficients of the polynomial a into register A and the first two coefficients of polynomial b into register B. It is important to note that only the upper word of Register B is visible to the processing units. By the end of CC3, the hardware has finished loading registers A and B, which corresponds to steps 6 and 7 in Algorithm 10. The control unit now begins the execution stage.

cycle	PU2	PU1	PU0
0	$reset$ = '1'		
	$C_2 = 0$ $Save_2 = 0$	$C_1 = 0$ $Save_1 = 0$	$C_0 = 0$ $Save_0 = 0$
1	$reset$ = '0'		
	$A_2 = 0$ $B_1 = 0$	$A_1 = 0$ $B_1 = 0$	$A_0 = a[2]$ $B_1 = 0$
2	$A_2 = 0$ $B_1 = b[0]$	$A_1 = a[2]$ $B_1 = b[0]$	$A_0 = a[1]$ $B_1 = b[0]$
3	$A_2 = a[2]$	$A_1 = a[1]$	$A_0 = a[0]$

Table 7.1: Initialization Stage

The execution stage for the first set of columns is outlined in Table 7.2. At CC4, the processing units (PUs) process their respective inputs and the control unit latches the output of each PU into register C. In addition, the next coefficient from polynomials a and b is loaded into their respective registers. The control unit repeats the operations that occurred in CC4 five $(N - 2)$ more times. Finally, at the end of CC9, all operations for this stage are completed. At CC10, the control unit begins execution of the load/write stage to set up the registers for the next set of three columns and to write the previous columns' results to $Cache\ C$.

cycle	PU2	PU1	PU0
4	$C_2 = C_2 + A_2 \cdot B_1$ $A_2 = a[1]$ $B_1 = b[1]$	$C_1 = C_1 + A_1 \cdot B_1$ $A_1 = a[0]$ $B_1 = b[1]$	$C_0 = C_0 + A_0 \cdot B_1$ $A_0 = a[6]$ $B_1 = b[1]$
5	$C_2 = C_2 + A_2 \cdot B_1$ $A_2 = a[0]$ $B_1 = b[2]$	$C_1 = C_1 + A_1 \cdot B_1$ $A_1 = a[6]$ $B_1 = b[2]$	$C_0 = C_0 + A_0 \cdot B_1$ $A_0 = a[5]$ $B_1 = b[2]$
6	$C_2 = C_2 + A_2 \cdot B_1$ $A_2 = a[6]$ $B_1 = b[3]$	$C_1 = C_1 + A_1 \cdot B_1$ $A_1 = a[5]$ $B_1 = b[3]$	$C_0 = C_0 + A_0 \cdot B_1$ $A_0 = a[4]$ $B_1 = b[3]$
7	$C_2 = C_2 + A_2 \cdot B_1$ $A_2 = a[5]$ $B_1 = b[4]$	$C_1 = C_1 + A_1 \cdot B_1$ $A_1 = a[4]$ $B_1 = b[4]$	$C_0 = C_0 + A_0 \cdot B_1$ $A_0 = a[3]$ $B_1 = b[4]$
8	$C_2 = C_2 + A_2 \cdot B_1$ $A_2 = a[4]$ $B_1 = b[5]$	$C_1 = C_1 + A_1 \cdot B_1$ $A_1 = a[3]$ $B_1 = b[5]$	$C_0 = C_0 + A_0 \cdot B_1$ $A_0 = a[2]$ $B_1 = b[5]$
9	$C_2 = C_2 + A_2 \cdot B_1$ $A_2 = a[3]$ $B_1 = b[6]$	$C_1 = C_1 + A_1 \cdot B_1$ $A_1 = a[2]$ $B_1 = b[6]$	$C_0 = C_0 + A_0 \cdot B_1$ $A_0 = a[1]$ $B_1 = b[6]$

Table 7.2: Execution Stage

Finally, the load/write stage is outlined in Table 7.3. Starting at CC10, the PUs process the last set of inputs from CC9 and the control unit:

- latches the output of each PU into register *Save* (Algorithm 10, Step 14),
- clears register C (Algorithm 10, Step 15),
- begins serially loading shift register A with the first three coefficients of the first row of the 2nd set of three columns shown in Figure 7.2, and
- loads a single coefficient into register B (Algorithm 10, Step 19).

For CC11 through CC13, the control unit writes the final result of the previous set of columns to *Cache C*. This is done by transmitting the least significant slot of the *Save* register ($Save_0$) to *Cache C* and enabling the register to perform an 8-bit shift right operation for each one of these clock cycles. By the end of CC12, the hardware

has finished loading register A, which corresponds to step 18 in Algorithm 10. During CC13, as the last coefficient from the $Save$ register is being written to $Cache\ C$, the control unit also begins re-execution of the execution stage for the second column set, which is followed by another execution of the load/write stage. Finally, the execution and load/write stages need to executed one more time to process the third (and last) column set and write its results to $Cache\ C$.

cycle	**PU2**	**PU1**	**PU0**
10	$Save_2 = C_2 + A_2 \cdot B_1$ $C_2 = 0$ $A_2 = a[2]$ $B_1 = b[0]$	$Save_1 = C_1 + A_1 \cdot B_1$ $C_1 = 0$ $A_1 = a[1]$ $B_1 = b[0]$	$Save_0 = C_0 + A_0 \cdot B_1$ $C_0 = 0$ $A_0 = a[5]$ $B_1 = b[0]$
11	$c_2 = Save_0$ $Save = (0, Save_2 \cdots Save_1)$		
	$A_2 = a[1]$	$A_1 = a[5]$	$A_0 = a[4]$
12	$c_1 = Save_0$ $Save = (0, Save_2 \cdots Save_1)$		
	$A_2 = a[5]$	$A_1 = a[4]$	$A_0 = a[3]$
13	$c_0 = Save_0$		

Table 7.3: Load/Write Stage

In summary, the initialization stage is only executed at the start of a polynomial multiplication. Then, the execution and load/write stages are executed $\lfloor \frac{N+u-1}{u} \rfloor$ times. Once the last load/write stage has completed, the control unit asserts the *done* signal to indicate to the host system that it has completed the polynomial multiplication.

7.3 Processing Unit

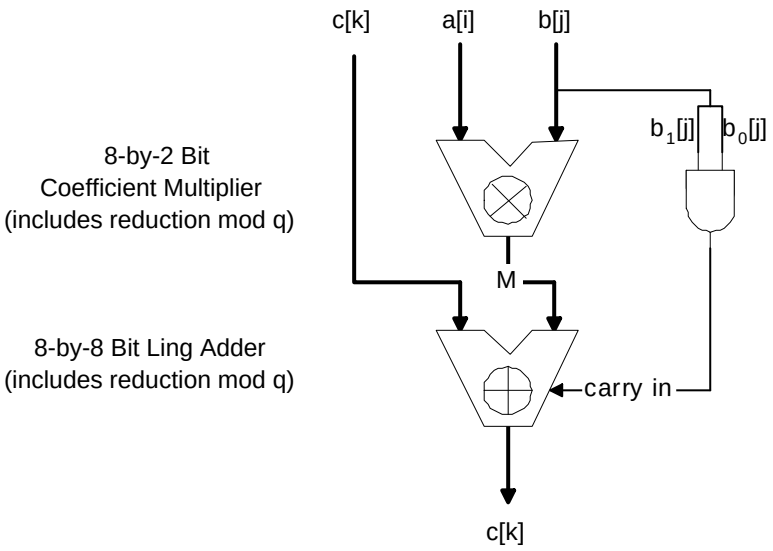


Figure 7.4: Processing Unit

The processing unit (PU), shown in Figure 7.4, is responsible for performing the operation in (7.1). It consists of a coefficient multiplier and a Ling adder [15], which is an improved carry look ahead adder [16]. Both of these components incorporate the reduction modulo q. The components of the processing unit consist solely of combinational logic and are not dependent upon a rising edge clock signal. The coefficient multiplier, shown in Figure 7.5, computes $M = a[i] \cdot b[j] \pmod{q}$ portion of (7.1). The main hardware consists of eight 2 by 1-bit multipliers, which can be identified as the boxes shown in Figure 7.5. Each of the 2 by 1-bit multipliers was designed to behave according to the truth table in Table 7.4. When referring to the truth table the following should be noted:

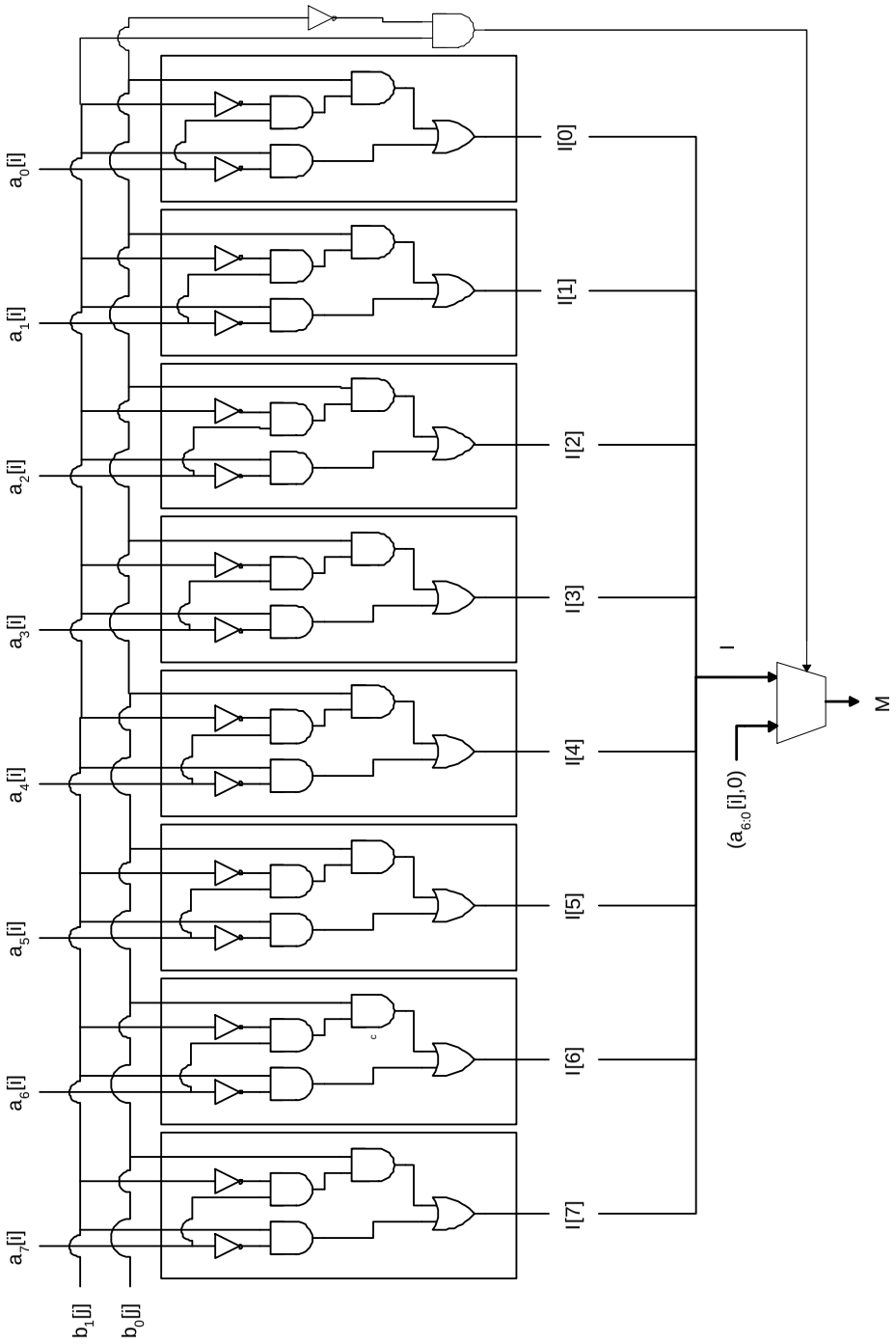


Figure 7.5: 8 by 2-bit Coefficient Multiplier

$a_i[i]$	$b_1[j]$	$b_0[j]$	I_i
0	0	0	0
0	0	1	0
0	1	0	d
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	d
1	1	1	0

Table 7.4: 2 by 1-bit Multiplier Truth Table

- The 2-bit representation for $b[j]$ is interpreted differently than its decimal conversion by this design as shown in Table 7.5.

bit representation	**integer representation**
00	0
01	1
10	2
11	-1

Table 7.5: Interpretation of b[j]'s binary bits

- For the case $b[j] = (11)_2 = -1$, it is necessary that $a[i]$ be converted to its two's complement representation in order for the Ling adder to subtract. However, this multiplier only inverts $a[i]$ to reduce the complexity of the design. The two's complement conversion is completed by setting the *carry in* of the Ling adder to '1'. This is accomplished by the AND gate shown in Figure 7.4.

- For the case $b[j] = (10)_2 = 2$, it should be noted that this operation is not performed by the main hardware of the multiplier as indicated by the don't care condition in Table 7.4. Hence, a multiplexer is needed to pass the left shifted value of $a[i]$ for when $b[j] = 2$, otherwise, I is passed to the output from the main hardware.

The reduction modulo q portion of the equation is handled by ignoring any carries that exceed the 8-bit boundary, which would only occur for the case $b[j] = 2$. Finally, the output of the multiplexer, M, is passed on to the 8-bit Ling adder for accumulation.

The 8-bit Ling Adder, in Figure 7.6a, is responsible for computing $c[k] = M + c[k] \pmod{q}$ of (7.1) and it was designed in a hierarchical manner similar to [17]. The 8-bit adder receives two 8-bit inputs, which are split in half, and a carry in. Half of each operand is then passed to one of the two 4-bit adders, which is further broken down into four partial full adders (PFA) and one 4-bit carry look ahead (CLA) network as shown in Figure 7.6b. This four bit adder is based on Ling's design discussed in [15, 18]. For this design, the PFA is responsible for generating the propagate (P), generate (G), and sum (S) signals for the 4-bit CLA network. The logic for these signals is expressed in Equations (7.2) through (7.4) and the schematic can be viewed in Figure 7.7a.

$$P = a_i[i] + b_i[j] \tag{7.2}$$

$$G = a_i[i] \cdot b_i[j] \tag{7.3}$$

$$S = a_i[i] \oplus b_i[j] \oplus carry \tag{7.4}$$

The function of the 4-bit CLA is to generate the carries for each of the PFAs. The four carry outputs, $C1$ through $C4$, are computed in hardware by the expressions shown in Equations (7.5) through (7.8). The schematic of the 4-bit CLA can be viewed in Figure 7.7b.

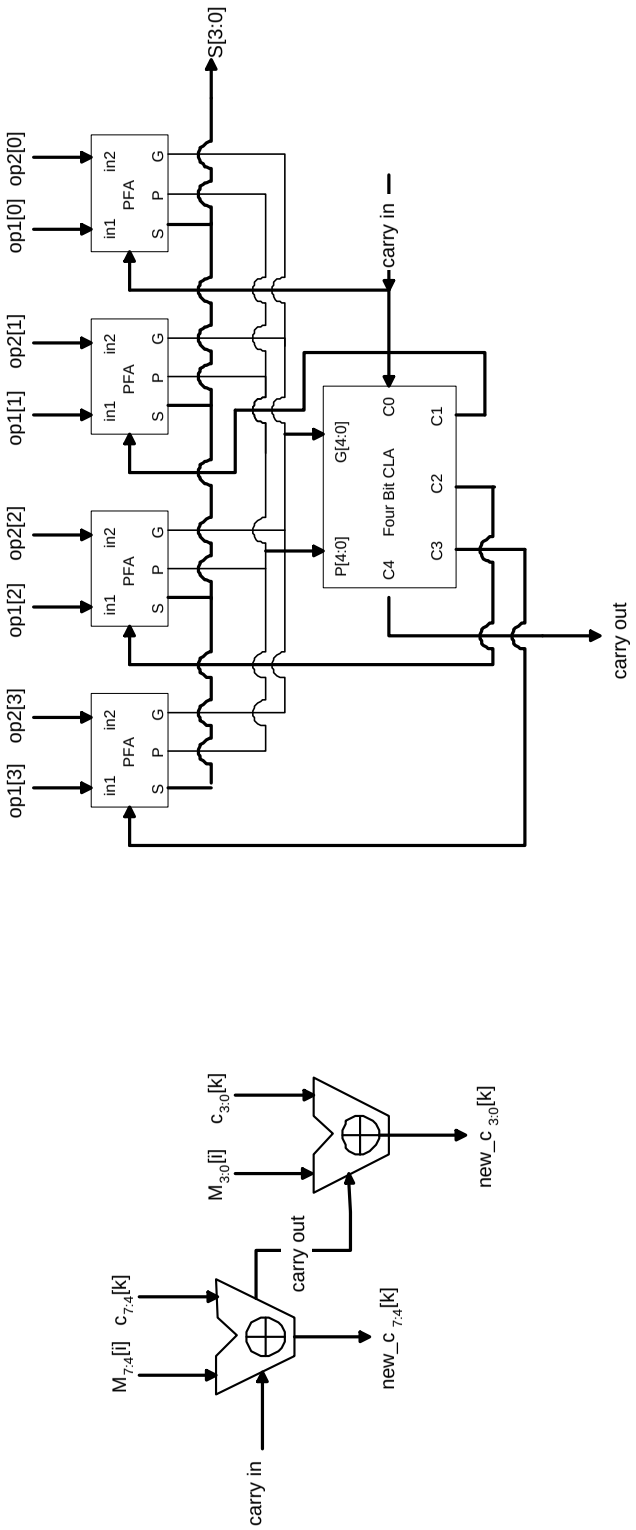


Figure 7.6: 8-bit Ling Adder (a) and 4-bit Ling Adder (b)

$$h_1 = G[0] + C0 \tag{7.5}$$
$$C1 = P[0] \cdot h_1$$

$$h_2 = G[1] + G[0] + (P[0] \cdot C0) \tag{7.6}$$
$$C2 = P[1] \cdot h_2$$

$$h_3 = G[2] + G[1] + (P[1] \cdot G[0]) + (P[1] \cdot P[0] \cdot C0) \tag{7.7}$$
$$C3 = P[2] \cdot h_3$$

$$h_4 = G[3] + G[2] + (P[2] \cdot G[1]) + (P[2] \cdot P[1] \cdot G[0]) + (P[2] \cdot P[1] \cdot P[0] \cdot C0) \tag{7.8}$$
$$C4 = P[3] \cdot h_4$$

Finally, the modulo q reduction is performed by simply ignoring the carry out of the 2^{nd} four bit adder, which received the most significant half of the operands. The output of this 8-bit adder produces the final result of the partial product operation shown in (7.1).

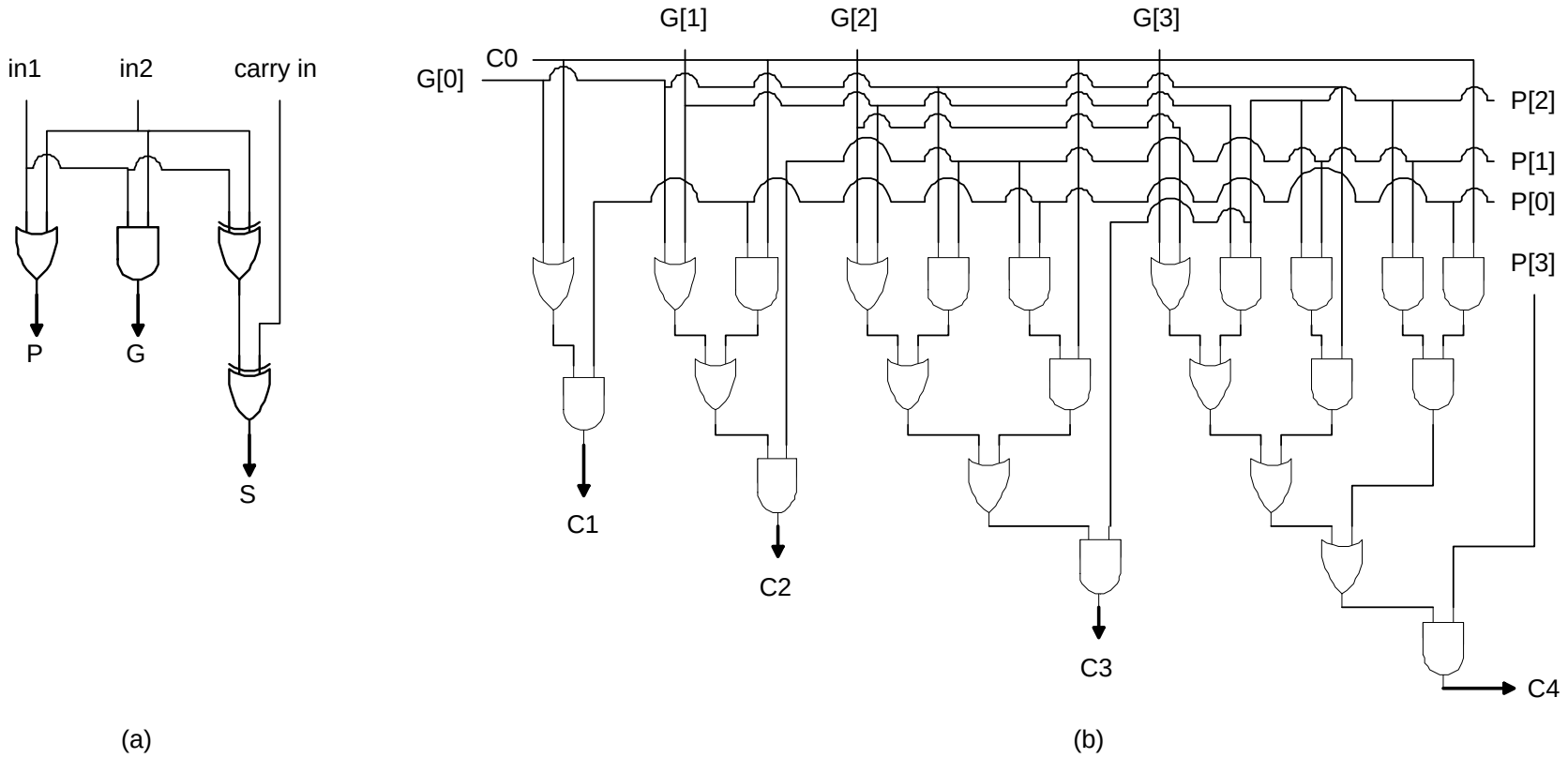


Figure 7.7: Partial Full Adder (a) and 4-bit CLA (b)

Chapter 8

Supported Operations and Limitations

The NTRU multiplier provides support for all the polynomial multiplications required by the public key creation, encryption, and decryption procedures of NTRU. As mentioned earlier, the integer moduli must be set to $p = 3$ and $q = 256$. The polynomial size N, however, can be set to arbitrary lengths depending on the security requirements. The following indicates which operations are supported in the public key creation, encryption, and decryption processes, respectively, as well as the assumptions associated with them.

1. **Public Key Creation:** $h = F_q * g \pmod{q}$

   The NTRU multiplier can perform the full operation above assuming that:

   - The random polynomial g has coefficients from $\{-1, 0, 1\}$, and

- The inverse polynomial F_q of the private key f modulo q has been pre-computed and has coefficient in the range $(0,\ q-1)$.

2. **Encryption:** $e = pr * h + m \pmod{q}$

   The NTRU multiplier can only perform the multiplication of $r * h$ (mod q). It is assumed that:

   - The random polynomial r has coefficients from $\{-1, 0, 1\}$,
   - The addition of the message m occurs outside of the multiplier and it has coefficients from $\{-1, 0, 1\}$,
   - 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 NTRU multiplier has no problems computing Step (a), while Step (b) is a simple operation that can be performed outside of the multiplier. However,

Step (c) cannot be computed by the multiplier because the polynomial a's coefficients are no longer unsigned. In addition, the multiplier does not support the reduction modulo p. Fortunately, there is a way to slightly modify the steps so that the NTRU multiplier can perform the polynomial multiplications within the decryption process. Four steps are now required:

(a) $a = f * e \pmod{q}$

The NTRU multiplier can perform the full operation above as long as the random polynomial, f, has coefficients from $\{-1, 0, 1\}$.

(b) Shift coefficients of a from $(0,\ q-1)$ to $(-\frac{q}{2},\ \frac{q}{2})$

Outside of the NTRU multiplier, the user needs to shift the coefficients of a from the range $(0,\ q-1)$ to $(-\frac{q}{2},\ \frac{q}{2})$.

(c) $b = a \pmod{p}$

Again, the NTRU 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 polynomial multiplication by the NTRU multiplier. However, the multiplier will only reduce the coefficients of d modulo q. Step (d) requires reduction modulo p, so, it is the responsibility of the user to reduce the coefficients of d modulo p in order to receive the correct message.

As mentioned earlier, the NTRU multiplier is capable of enhancing the performance of the polynomial multiplications, whose coefficients need to be reduced modulo q. However, the multiplier is not capable of reducing the coefficients modulo $p = 3$, which outlines the NTRU multiplier's limitation. The user can implement a modulo $p = 3$ reduction circuit outside of the NTRU multiplier at a cost of approximately 50 gates.

Chapter 9

Performance Analysis

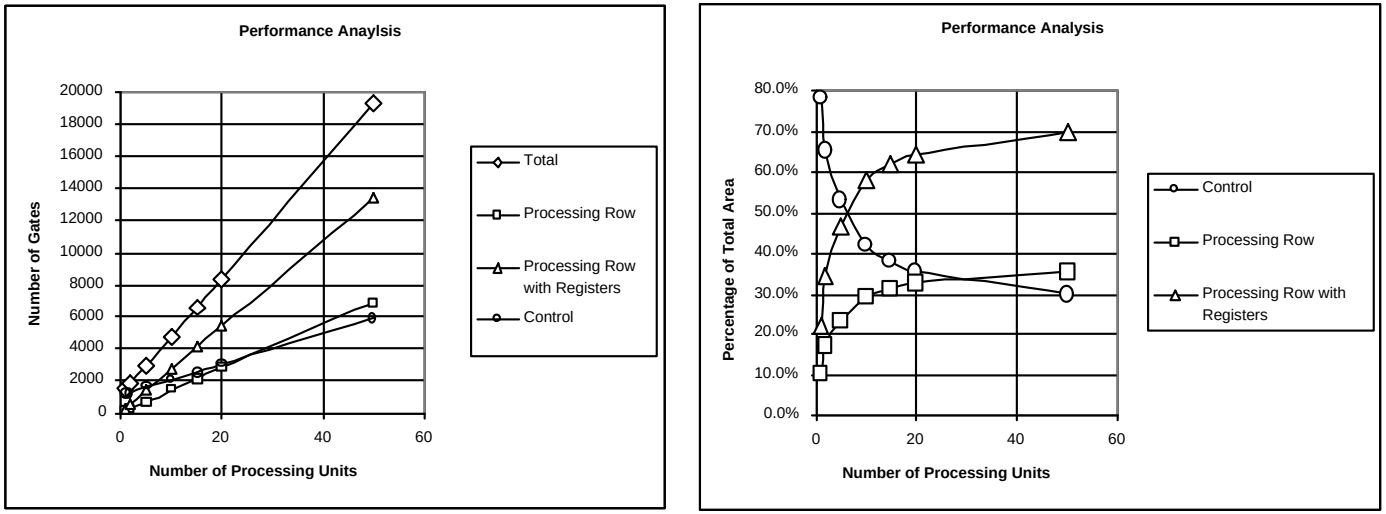
This section summarizes the performance of the NTRU multiplier for a range of processing units. Since this is the first realization of a hardare implementation that soley performs a polynomial multiplication for NTRU, no comparison on the performance of this design can be made. To demonstrate the worst case scenario, the performance analysis was conducted for NTRU's highest security level ($N = 503$). The data in Table 9.1 summarizes the overall numerical results for various performance criteria. The design was modeled using VHDL, simulated for functionality using Mentor Graphics' ModelSim 5.5f, and synthesized with Mentor Graphics' LeonardoSpectrum tool using fast TSMC 0.35μ technology [19].

# of processing units	1	2	5	10	15	20	50
# gates A's shift reg	70	129	307	604	900	1196	2975
# gates B's shift reg	33	33	33	33	33	33	33
# gates C's reg	70	140	350	700	1050	1400	3500
# gates total for regs	173	302	690	1337	1983	2629	6508
# gates for processing row	154	300	684	1379	2065	2754	6907
# gates for control	1156	1148	1558	1979	2503	3003	5855
# gates total	1483	1750	2932	4695	6551	8386	19270
% processing row	10.4%	17.1%	23.3%	29.4%	31.5%	32.8%	35.8%
% processing row with regs	22.0%	34.4%	46.9%	57.8%	61.8%	64.2%	69.6%
% control	78.0%	65.6%	53.1%	42.2%	38.2%	35.8%	30.4%
Frequency (MHz)	290.9	304.4	201.2	200.2	196.1	197.4	194.4
Clock Period (ns)	3.44	3.29	4.97	5.00	5.10	5.07	5.14
Time for 1 polynomial mult (ms)	0.878	0.421	0.255	0.129	0.088	0.067	0.029
Speed up	1	2.1	3.4	6.8	10.0	13.1	30.5

Table 9.1: Performance Analysis for Optimized and Scalable Design

As expected, the area increases at a linear rate which is seen in Figure 9.1a. It is interesting to note that as the size of the processing row gets larger, a smaller percentage of the area is spent on control and the majority of the area is spent on the main system, which includes the processing row and registers. This behavior, as shown in Figure 9.1b, occurs because the only elements that require additional hardware in the control are the relatively small counters so it increases at a slower rate than the main system. Whereas, the main system increases at a much larger rate due to multiple replications of the processing unit.

Although the area increases linearly with the size of the processing row, the performance of the NTRU multiplier increases linearly as well. This multiplier can be clocked at a maximum of 290.9MHz for one processing unit and 194.4MHz for fifty



(a) (b)

Figure 9.1: Gate Count Distribution (a) and Percentage of total area (b)

processing units. For this design, the counters are the major bottleneck. Therefore, as the number of processing units increases, the counters cause a small decline in frequency. From the data in Table 9.1, it is also noticeable that the frequency tends to fluctuate. This is due to the optimizations performed by the synthesis tool. With these measured frequencies shown in Table 9.1, the NTRU multiplier can complete a polynomial multiplication in 0.9 ms for one processing unit and in only 29 μs for fifty processing units. This scalable architecture greatly benefits the performance of NTRU since it provides a speed up that increases linearly with the number of processing units as shown in Figure 9.2.

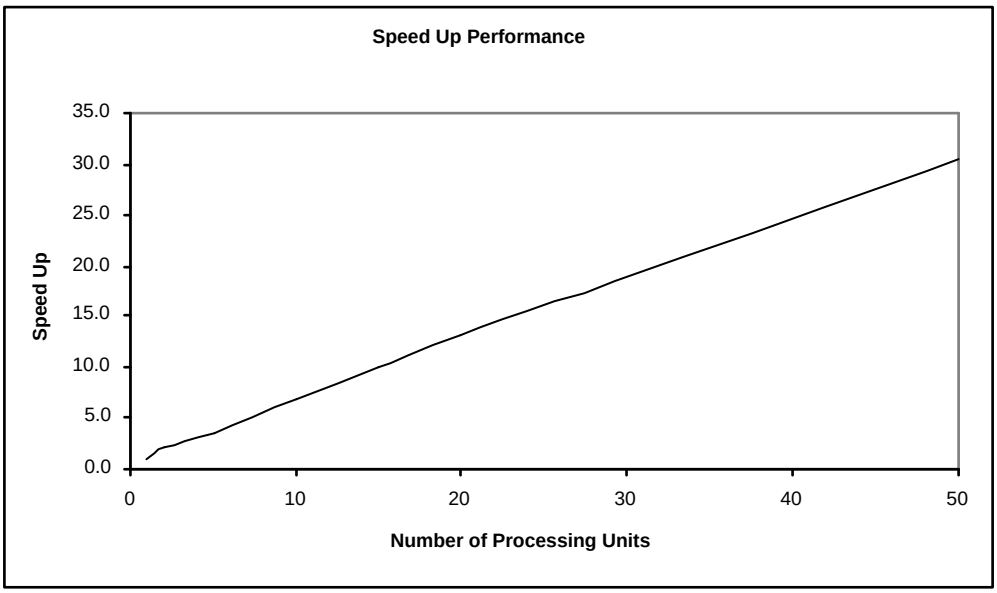


Figure 9.2: Speed up distribution

Part III

Unified NTRU & Montgomery Multiplier

Chapter 10

Previous Work

This section will review some previous and current contributions to the design of unified architectures. Although these contributions do not support NTRU, they do propose unified designs for performing arithmetic in both finite fields, $GF(p)$ and $GF(2^k)$.

The work in [20] introduces a scalable and unified multiplier that uses the Montgomery Multiplication algorithm to compute modular multiplications for fields $GF(p)$ and $GF(2^k)$. This design supports arithmetic in both fields through the introduction of the "dual-field adder", which performs addition with and without carry propagation. In order to avoid designing a dual field multiplier, this work utilizes a word-level bit-serial version of the Montgomery Multiplication algorithm. Therefore, only a bit-serial multiplier is necessary in which the multiplier is fed bit-serially and the multiplicand is fed in word size chunks. Scalability in this design is achieved by in-

creasing the number of fixed area multiplier units so that any operand size can be supported. In addition, the pipeline depth and word size can be configured to meet the desired area and performance specifications.

Großschädl's paper [21] designs a unified multiplier without using Montgomery Multiplication. Instead, this work utilizes a MSB-first shift-and-add method [22] for the operations in both fields $GF(p)$ and $GF(2^k)$. For this design, one operand is scheduled bit serially while the other operand is scheduled fully parallel, in which the full n-bit integer is fed into the system. Therefore, this design is not scalable or reconfigurable.

Gaubatz has designed a high-radix scalable unified Montgomery Multiplier [23] (a Master's Thesis completed May 1, 2002). Similar to Savaş et al. [20], Gaubatz's design is based on the word-level Montgomery Multiplication algorithm. In contrast, this Montgomery Multiplier is high-radix since it has a dual field multiplier that supports $w \times w$-bit multiplications. In addition, his design utilizes the addition tree method to reduce the delay path to $\log \frac{3}{2}w$ (instead of w) in his dual field multiplier and adder. The word size of the multiplier core and the pipelining depth can be configured before implementation. Also, Gaubatz's design is scalable in terms of operand size, which is achieved through multiple mappings of the multiplier core.

Chapter 11

Montgomery Multiplication

Montgomery Multiplication was introduced by P.L. Montgomery in [24] to improve the performance of computing the modular multiplication shown below:

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

where a, b, c, and m are integers. The reduction modulo m requires a division which is very costly to do in hardware. Instead, Montgomery replaces this division with a series of shift operations, which is very simple to perform in hardware. Montgomery's technique requires that each operand be converted into its residue representation by multiplying it with an integer residue, R, as shown below:

$$\overline{a} = a \cdot R \pmod{m} \tag{11.1}$$

$$\overline{b} = b \cdot R \pmod{m}.$$

There are two restrictions on choosing a residue, R:

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

After computation of the residues, the Montgomery Multiplication algorithm is executed to compute the product, c:

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

As indicated by the bar notation above, the result, c, is in residue form. To convert the result back to non-residue format, the Montgomery Multiplication algorithm needs to be executed again as follows:

$$\begin{aligned} MM(\overline{c}, 1) &= \overline{c} \cdot 1 \cdot R^{-1} \pmod{m} \\ &= c \cdot R \cdot 1 \cdot R^{-1} \pmod{m} \\ &= c \pmod{m} \end{aligned}$$

Since the initial residue computations in (11.1) require divisions, using this Montgomery algorithm to perform a single modular multiplication will not provide any benefits over the straightforward method. Ideally, Montgomery's technique should be used for applications that require multiple modular multiplications over the same modulus (e.g. modular exponentiations) because the residue will be preserved for

each consecutive multiplication. Therefore, the transformation to and from residue format is only necessary at the beginning and end of the chain of multiplications.

11.1 Word-Level Montgomery Multiplication Algorithm

The word-level Montgomery Multiplication algorithm for $GF(p)$ is given in Algorithm 11. The algorithm is a representative of several algorithms that were reviewed in [25]. Within this algorithm:

- $a[i]$, $b[j]$, $c[i]$, and $m[i]$ represent individual words of the word vector representations of the long integers a, b, c, and m, respectively,
- the least significant word is denoted by the 0^{th} element of the vector (e.g. $m[0]$),
- the word size is w bits in length,
- n is the number of words in the vectors,
- $m[0]' = -m[0]^{-1} \bmod 2^w$ 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 11, a long subtraction of $c = c - m$ may be necessary if the result, c, is larger than the modulus, m.

Algorithm 11 Word-Level Montgomery Algorithm for $GF(p)$

1: **for** $j = 0$ to $n - 1$ **do**
2: $CS = (a[0] \cdot b[j]) + c[0]$
3: $U = S \cdot m[0]' \bmod 2^w$
4: $CS = CS + (m[0] \cdot U)$
5: $CS \gg w$
6: **for** $i = 1$ to $n - 1$ **do**
7: $CS = CS + (a[i] \cdot b[j]) + (m[i] \cdot U) + c[i]$
8: $c[i - 1] = S$
9: $CS \gg w$
10: **end for**
11: $c[n - 1] = S$
12: **end for**

With a few modifications as proposed in [26], Algorithm 11 can be used for polynomials in $GF(2^k)$ as well. The modifications, shown in Algorithm 12, include changing integer multiplications to polynomial multiplications ($\odot$) and additions to word size XOR operations ($\oplus$). 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.

Algorithm 12 Word-Level Montgomery Algorithm for $GF(2^k)$

1: **for** $j = 0$ to $n - 1$ **do**
2: $CS = (a[0] \odot b[j]) \oplus c[0]$
3: $U = S \odot m[0]' \bmod 2^w$
4: $CS = CS \oplus (m[0] \odot U)$
5: $CS \gg w$
6: **for** $i = 1$ to $n - 1$ **do**
7: $CS = CS \oplus (a[i] \odot b[j]) \oplus (m[i] \odot U) \oplus c[i]$
8: $c[i - 1] = S$
9: $CS \gg w$
10: **end for**
11: $c[n - 1] = S$
12: **end for**

Chapter 12

High-Radix Montgomery Multiplier Core

Gaubatz's high-radix Montgomery Multiplier Core [23] shown in Figure 12.1 is capable of performing all operations required by the Montgomery Multiplication algorithm for both fields $GF(p)$ and $GF(2^k)$. The core consists of two $w \times w$-bit dual field multipliers, a $w \times w$-bit dual field adder, and a three-way dual field adder. The three-way dual field adder accepts two operands that are $2w$ bits in length and a third operand that is $w+1$ bits in length. The field in which the dual field components will operate is determined by $\mathbf{f\_sel_0}$. If $\mathbf{f\_sel_0} = 1$, the units will perform $GF(p)$ arithmetic. Otherwise, if $\mathbf{f\_sel_0} = 0$, the units perform $GF(2^k)$ arithmetic. The following describes how each major step of the word-level Montgomery Multiplication algorithm is supported by the core. Since $\mathbf{f\_sel_0}$ selects between the two fields, Algorithm 11 and 12 are es-

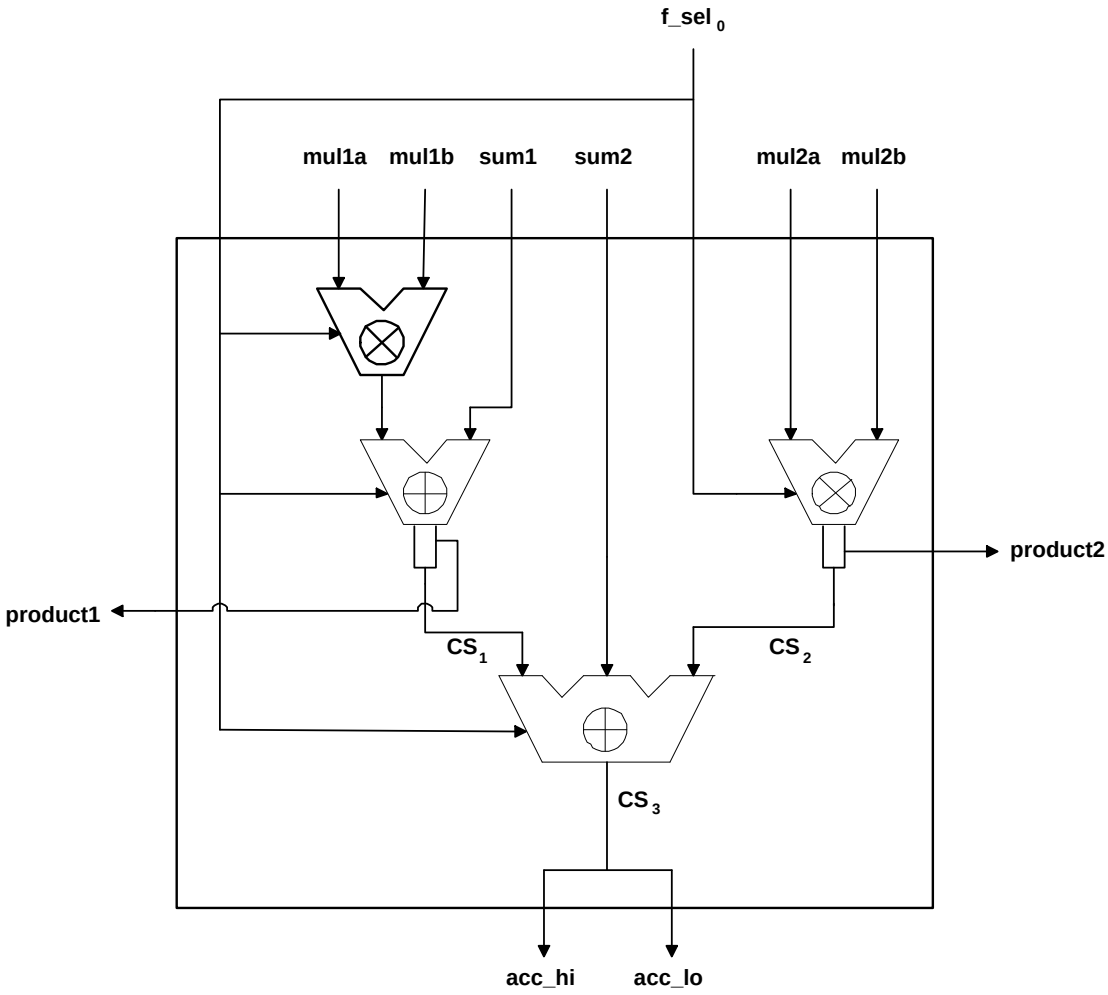


Figure 12.1: Gaubatz's Montgomery Multiplier Core

sentially the same algorithm to the core. So, for the description below, it is adequate to only reference Algorithm 11. Also, the steps consisting of the for loop statements are not described below because they are handled by a control source outside of the core not by the core itself. As a note, the $\Leftarrow$ and $\Rightarrow$ represent the assignment of the input/output ports of the core by the control source to the variables in Algorithm 11.

Step 2: $CS = (a[0] \cdot b[j]) + c[0]$

Since the operation above requires a multiplication and an addition, the first multiplier is used. The inputs of the core are assigned as follows:

- $mul1a \Leftarrow a[0]$
- $mul1b \Leftarrow b[j]$
- $sum1 \Leftarrow c[0]$

After the multiplier and adder have finished processing the inputs, the $2w$-bit result CS in Step 2 is CS_1 in Figure 12.1.

Step 3: $U = S \cdot m[0]' \bmod 2^w$

Since the only operation required in Step 3 is a multiplication, the second multiplier is used. The inputs and output of the core are assigned as follows:

- $mul2a \Leftarrow product1$
- $mul2b \Leftarrow m[0]'$
- $product2 \Rightarrow U$

The S in Step 3 is the lower word of the result, CS_1, of Step 2. This lower word of CS_1, which is w bits in length, is transmitted out of the core via $product1$. So, this is the reason $product1$ is assigned to $mul2a$ of the second multiplier. Also, the equation in Step 3 requires the $2w$-bit product of $S \cdot m[0]'$ (CS_2 in Figure 12.1) be reduced modulo 2^w. The reduction modulo 2^w is simply the extraction of the lower word of CS_2, which is transmitted out of the core as $product2$. Since $product2$ will need to be accessible many clock cycles later and the second multiplier will need to used again, $prodcut2$ is assigned to a variable U.

Step 4: $CS = CS + (m[0] \cdot U)$

The operation in Step 4 requires a $w \times w$-bit multiplication then a $2w \times 2w$-bit addition. As a result, the second multiplier and the three-way adder are used for the computation of Step 4. The inputs are assigned as follows:

- $mul2a \Leftarrow m[0]$
- $mul2b \Leftarrow U$
- $sum2 \Leftarrow 0$

As a note, the result from the operation in Step 2 (CS_1), the result of $m[0] \cdot U$ (CS_2), and $sum2$ 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: $CS = CS \gg w$ **and Step 7:** $CS = CS + (a[i] \cdot b[j]) + (m[i] \cdot U) + c[i]$

For these steps, all of the components of the core are used. So, the inputs are assigned

as follows:

- $mul1a \Leftarrow a[i]$
- $mul1b \Leftarrow b[j]$
- $mul2a \Leftarrow m[i]$
- $mul2b \Leftarrow U$
- $sum1 \Leftarrow c[i]$
- $sum2 \Leftarrow acc\_hi$

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 $w+1$ bits, acc_hi, of previous operation in Step 4 or Step 7, respectively. 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.

Step 8: $c[i-1] = S$

- $acc\_lo \Rightarrow c[i-1]$

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 of the

Montgomery Multiplication.

Step 11: $c[n-1] = S$

- $acc\_hi_{w-1\cdots 0} \Rightarrow c[n-1]$

Basically, 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. As mentioned earlier, the shift operation in Step 9 can be eliminated by just assigning the necessary portion of the upper word of the result from Step 7, acc_hi. Since the lower word, S, is w bits, only the least significant w bits of acc_hi is assigned to the output.

Chapter 13

Hardware Design

This work focuses on designing an unified architecture using Montgomery Multiplication to perform a modular multiplication for finite fields, $GF(p)$ and $GF(2^k)$, and a polynomial multiplication for NTRU. For this design, the integer moduli are fixed to $p = 3$ and $q = 256 = 2^8$ and the coefficients of the product polynomial for NTRU are reduced modulo q. The following paragraphs will detail the required conditions that need to be met in order for the Montgomery Multiplication algorithm to work for all three cases.

For the application of NTRU to the Montgomery Multiplication algorithm, setting the residue, $R = x^N$, introduces the unique property shown below:

Since the modulus:

$$\begin{aligned} x^N - 1 &\equiv 0 \pmod{x^N - 1} \\ x^N &\equiv 1 \pmod{x^N - 1} \end{aligned}$$

Then:

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

So, when this property is applied to Montgomery's method, the residue of an operand is simply the operand itself as seen below

$$\begin{aligned} \overline{b} &= b \cdot R \pmod{x^N - 1} \\ &= b \cdot x^N \pmod{x^N - 1} \\ &= b \cdot 1 \pmod{x^N - 1} \\ &= b \pmod{x^N - 1} \end{aligned}$$

and the Montgomery product is in non-residue format as shown below

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

Therefore, NTRU's operands never need to be converted to and from residue format to receive the correct result from the Montgomery Multiplication algorithm. However, other changes are necessary within the algorithm. Since NTRU's modulus has $N + 1$ words, the inner loop in Algorithm 11 needs to be incremented by one to process all words of the modulus. Yet, the outer loop does not need to be modified since NTRU's operands only has N words. Therefore, only N shifts are needed, which is already defined within the algorithm. The new algorithm shown in Algorithm 13 includes the change mentioned above. In addition, the parameter n in Algorithm 11 is changed to N in Algorithm 13 in order to gain a clear understanding of how NTRU affects the Montgomery Multiplication algorithm.

Algorithm 13 Word-Level Montgomery Algorithm for $GF(p)$, $GF(2^k)$, and NTRU

```
1: for j = 0 to N − 1 do
2:    CS = (a[0] · b[j]) + c[0]
3:    U = S · m[0]′ mod 2^w
4:    CS = CS + (m[0] · U)
5:    CS ≫ w
6:    for i = 1 to N do
7:       CS = CS + (a[i] · b[j]) + (m[i] · U) + c[i]
8:       c[i − 1] = S
9:       CS ≫ w
10:   end for
11:   c[N] = S
12: end for
```

In consequence to increasing the length of the inner loop, the most significant word(s) of NTRU's operands will need to be padded with zeroes to match the length of the modulus. Also as a result of this modification, some restrictions and changes affect how the Montgomery Multiplication algorithm operates for the $GF(p)$ and $GF(2^k)$ cases as indicated below. If w represents the word size in bits and N represents the number of words of NTRU's operands, then the following restrictions must be met:

For $GF(p)$:

$$\begin{aligned} R &\geq 2^{N \cdot w} \quad (residue) \\ m &< 2^{N \cdot w} \quad (modulus) \end{aligned}$$

For $GF(2^k)$:

$$\begin{aligned} R(x) &\geq x^{N \cdot w} \quad (residue) \\ degree_{max}(m(x)) &\leq x^{N \cdot w} \quad (modulus) \end{aligned}$$

In addition, since the length of the inner loop was incremented by one, then the most significant word(s) of the operands for both cases, $GF(p)$ and $GF(2^k)$, and the

modulus for $GF(p)$ need to be padded with zeroes to match length of inner loop. Also, this applies to the modulus of the $GF(2^k)$ case only if the modulus is $< 2^{N \cdot w}$. With these new requirements, $GF(p)$ and $GF(2^k)$ will work with the algorithm presented in Algorithm 13.

There is one additional change that is necessary to make NTRU's polynomial multiplication work with the Montgomery Multiplication algorithm. Since the integer multiplications within the polynomial multiplication are reduced modulo $q = 2^8$, there are no carries from one partial product column to the next like the $GF(p)$ case. So, the upper word of the result from Step 4 and Step 7 in Algorithm 13 needs to be cleared before it is re-processed in Step 7. The details of how these changes are applied to the hardware are discussed in the next section.

13.1 Unified Architecture

Since NTRU's operands and modulus are polynomials whose coefficients are 8-bit integers, a high-radix and word level design is the most convenient. By extending Gaubatz's high-radix word-level Montgomery Multiplier core, this unified architecture provides support for NTRU with minimal change to the hardware as shown in Figure 13.1. Before implementation, the Montgomery Multiplier core is fixed for a word size $w = 8$ bits to support the maximum length of NTRU's polynomial coefficients. The required changes mentioned in the previous section do not require any hardware modifications to the core. Instead, these changes require either a simple

change to the control logic or additional hardware outside the core.

The first modification requires that the length of the inner loop in Algorithm 11 be incremented by one as shown in Algorithm 13. Although this modification requires the control logic to perform an additional iteration, this does not necessarily mean that the size of the counter has to change. For instance, changing the control to count from 500 to 501 still requires a 9-bit counter. So, no additional hardware on top of the original control (no NTRU support) is required to perform this additional loop. However, there will be cases where the size of the counter does need to change (e.g. incrementing the count from 511 to 512) and this will require a very small increase in the hardware. Yet, this can be avoided by careful selection of the operand size. Therefore, this modification can be considered irrelevant in terms of affecting the hardware but it is well worth noting for the awareness of the user.

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 which function the unified architecture needs to support, a two-bit f_sel signal is necessary. The least significant bit of $\mathbf{f\_sel}$, $\mathbf{f\_sel_0}$, determines whether the multipliers and adders perform integer multiplications and additions ($\mathbf{f\_sel_0} = 1$) or polynomial multiplications and word-size XOR operations ($\mathbf{f\_sel_0} = 0$). The most significant bit of $\mathbf{f\_sel}$, $\mathbf{f\_sel_1}$, determines whether NTRU is the selected (NTRU $\Rightarrow$ $\mathbf{f\_sel_1} = 1$, No NTRU $\Rightarrow$ $\mathbf{f\_sel_1} = 0$). Since, NTRU's polynomial multiplication relies on the

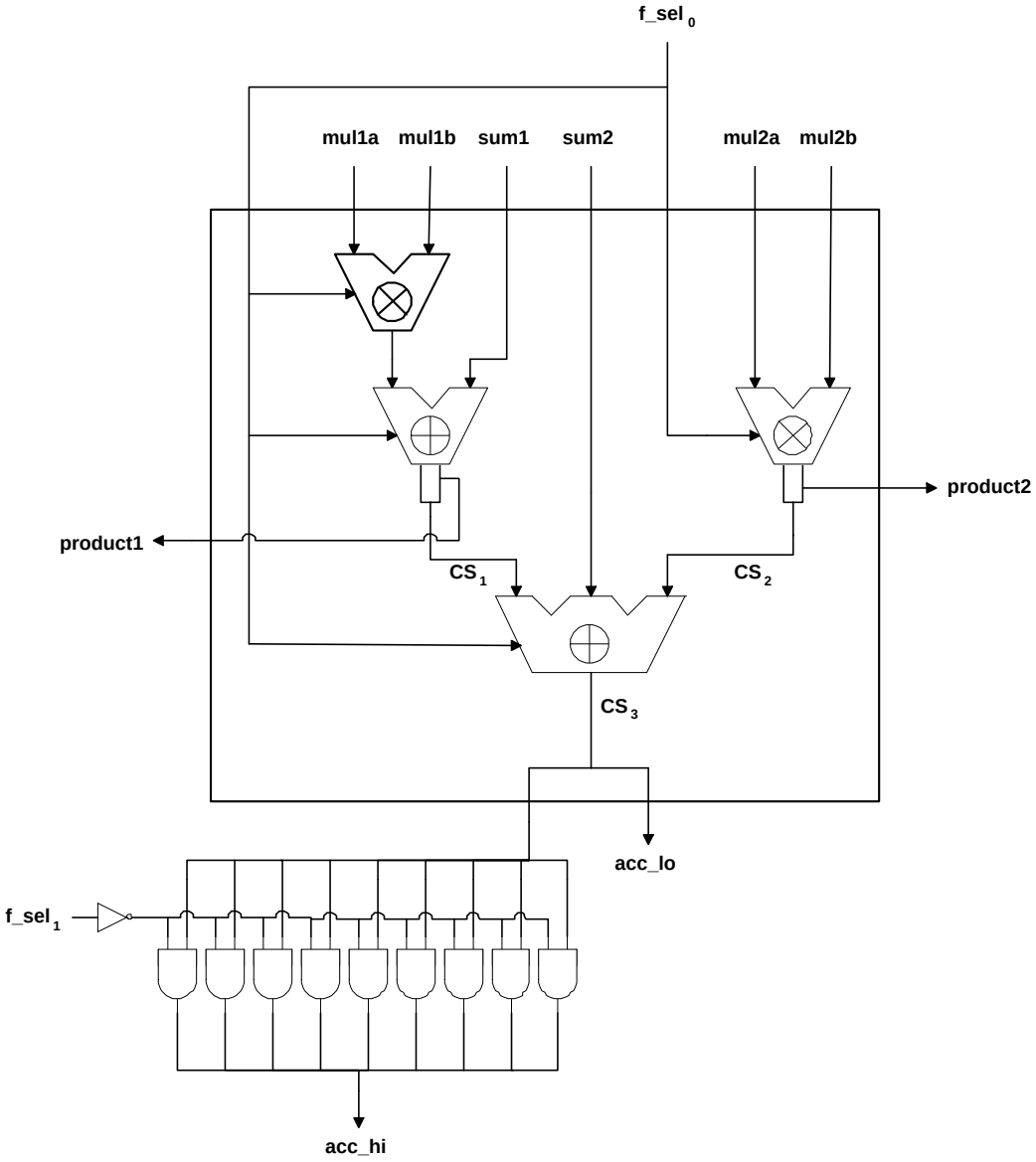


Figure 13.1: NTRU & Montgomery Multiplier

integer multiplication and addition of its coefficients, $\mathbf{f\_sel_0}$ needs to be set to '1' as well. Refer to Table 13.1 for clarification on the assignment of the **f_sel** signal and its selected function.

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

Table 13.1: Assignment of the f_sel signal

Now that a way to determine when NTRU is selected by user has been established, this same signal is used to clear the "carry word" as follows. The "carry word", acc_hi, is cleared by "AND-ing" each bit with the inverted $\mathbf{f\_sel_1}$ signal as shown in Figure 13.1. When NTRU is not selected, $\mathbf{f\_sel_1} = 0$, then after inversion $\mathbf{f\_sel_1}$ is set to '1'. As a result, all 9 bits of acc_hi are passed through the AND gates unchanged. However, if NTRU is selected, $\mathbf{f\_sel_1} = 1$, then after inversion $\mathbf{f\_sel_1}$ is set to '0'. In consequence, the AND gates zero out all 9 bits of acc_hi just as needed. Therefore, NTRU can be supported using the Montgomery Multiplier core with the addition of just 10 gates.

13.2 Control

The control assumes that all data necessary within Algorithm 13 is precomputed outside of the unified architecture before it is initialized to begin processing. Also, the control assumes that the word vectors a, b, c, and m reside in separate memory

caches. By using the Montgomery Multiplier core and the additional hardware shown in Figure 13.1, the control executes Algorithm 13 in seven stages. The host system initializes the unified architecture for processing by asserting the *reset* signal. At this point, Algorithm 13 is set up by the control via Stage 0. Then, the outer loop of Algorithm 13 is performed by executing Stages 1-6 N times. Finally, the inner loop in Algorithm 13 is performed by executing Stage 4-5 N times. The stages and their associated operations are explained in detail below. As a note when reviewing the stages, the control assumes that the word vectors a, b, c, and m reside in separate memory caches and the core executes the steps of the algorithm the same way as explained in Section 11.1.

Stage 0: **Setup**

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: **Step 2:** $CS = (a[0] \cdot b[j]) + c[0]$

Stage 1 is responsible for setting up the core to perform the operation in Step 2 as follows:

- $mul1a \Leftarrow a[0]$
- $mul1b \Leftarrow b[j]$
- $sum1 \Leftarrow c[0]$

Since Stage 2 does not require any new data from memory and $m[0]'$ is precomputed and stored, this stage does not need to transmit any new addresses to memory.

Stage 2: **Step 3:** $U = S \cdot m[0]' \bmod 2^8$

Stage 2 is responsible for setting up the core to perform the operation in Step 3 as follows::

- $mul2a \Leftarrow product1$
- $mul2b \Leftarrow m[0]'$

In addition to setting up the core, this stage transmits the address of $m[0]$ to memory so the data will be available for processing in the next stage.

Stage 3: **Step 4:** $CS = CS + (m[0] \cdot U)$

Stage 3 is responsible for setting up the core to perform the operation in Step 4 as follows:

- $mul2a \Leftarrow m[0]$
- $mul2b \Leftarrow product2$
- $sum2 \Leftarrow 0$

As mentioned in Section 11.1, $product2$ needs to be accessible many clock cycles later and the second multiplier will need to used again. So, $prodcut2$ is assigned to a variable U. For this stage, $mul2b$ is assigned $product2$ instead of U so that the data is available during this clock cycle. In addition to setting up the core, this stage

increments the index i and transmits the new addresses for a[i], b[j], m[i], and c[i] so that the data will be available for Stage 4.

Stage 4: **Step 7:** $CS = CS + (a[i] \cdot b[j]) + (m[i] \cdot U) + c[i]$

As mentioned in Section 11.1, the shift operations for Step 5 and Step 9 are eliminated by using the upper word result, *acc_hi*, instead of the lower word, *acc_lo*. This stage is responsible for setting up the core to perform the operation in Step 7 as follows:

- $mul1a \Leftarrow a[i]$
- $mul1b \Leftarrow b[j]$
- $mul2a \Leftarrow m[i]$
- $mul2b \Leftarrow U$
- $sum1 \Leftarrow c[i]$
- $sum2 \Leftarrow acc\_hi$

For clarification, $sum2$ now receives the result of *acc_hi* after it has passed through the AND gates. Therefore, *acc_hi* will be cleared if NTRU is selected. Also, this stage prepares for the write operation in Stage 5 by transmitting the address for $c[i-1]$.

Stage 5: **Step 8:** $c[i-1] = S$

- $acc\_lo \Rightarrow c[i-1]$

In addition to writing the lower word of CS_3 to the respective word location of c, this stage does one of the following two things. If the $for\ loop$ has not completed,

then this stage increments the index i and transmits the addresses for $a[i]$, $b[j]$, $c[i]$, and $m[i]$ in preparation for re-execution of Stage 4. Otherwise, this stage does not increment i and just transmits the address for $c[i]$ for Stage 6.

Stage 6: **Step 11:** $c[N] = S$

- $acc\_hi_{8-1\cdots 0} \Rightarrow c[N]$

Stage 6 writes the lower 8 bits of acc_hi to the N^{th} element in the word vector c. Also, this stage prepares for the re-execution of the outer loop by incrementing the index j, re-setting the index i back to zero, and transmitting the addresses for a[i], b[j], c[i].

Chapter 14

Performance Analysis

This section summarizes the performance of this unified multiplier for a range of modulus lengths. The data in Table 14.1 summarizes the overall numerical results for various performance criteria. As a reminder, N defines the max length of the modulus supported for $GF(p)$ and $GF(2^k)$ and the degree of the modulus polynomial for NTRU. This design was modeled using VHDL, simulated for functionality using Mentor Graphics' ModelSim 5.5f, and synthesized with Mentor Graphics' LeonardoSpectrum tool using TSMC 0.35μ technology [19].

N:	1	20	100	128	200	300	400	500	600
# bits per word (w):	8	8	8	8	8	8	8	8	8
# gates for control:	440	636	761	817	809	868	864	878	911
# gates added for NTRU support:	10	10	10	10	10	10	10	10	10
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
Equation for # clock cycles (CC):	$(2N+4)N+1$								
# CC for one unified multiplication:	7	881	20401	33281	80801	181201	321601	502001	722401
Time for one unifed multiplication (ms):	0.00	0.01	0.18	0.25	0.79	1.38	3.65	6.08	9.01

Table 14.1: Performance Analysis for Unified Design

As expected, the total area increases as the size of the modulus grows as seen in Figure 14.1a. This increase in the total area is due to the increase in the control logic. As the supported modulus length grows, the size of the counters within the control grows as well. Despite this increase, the area scales at a slow rate as shown in Figure 14.1a. Although the area dedicated to control increases with the modulus length, the majority of the area is used for the core as seen in Figure 14.1b, which is very ideal since it is the main processing hardware of the system.

In order to determine the performance of this unified multiplier for various supported modulus lengths, several timing estimations were obtained. Table 14.1 outlines the maximum frequency the system can be clocked for each measured modulus length. Due to the increase in control logic, the performance of the multiplier decreases as the size of the modulus grows. The number of clock cycles (#CC) for this architecture to complete a single unified multiplication is determined by the following equation:

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

Then, the time for the system to complete a unified multiplication is computed by multiplying the number of clock cycles with the respective clock period. The timing results shown in Table 14.1 indicate that the unified multiplier can complete a single unified multiplication for $N = 1$ in 50 ns and for $N = 600$ in a little over 9 ms. In addition, the performance of this unified multiplier was estimated for three different applications representing each function the multiplier can now support in Table 14.2. For the $GF(p)$ case, the unified multiplier would compute a 1024-bit RSA operation

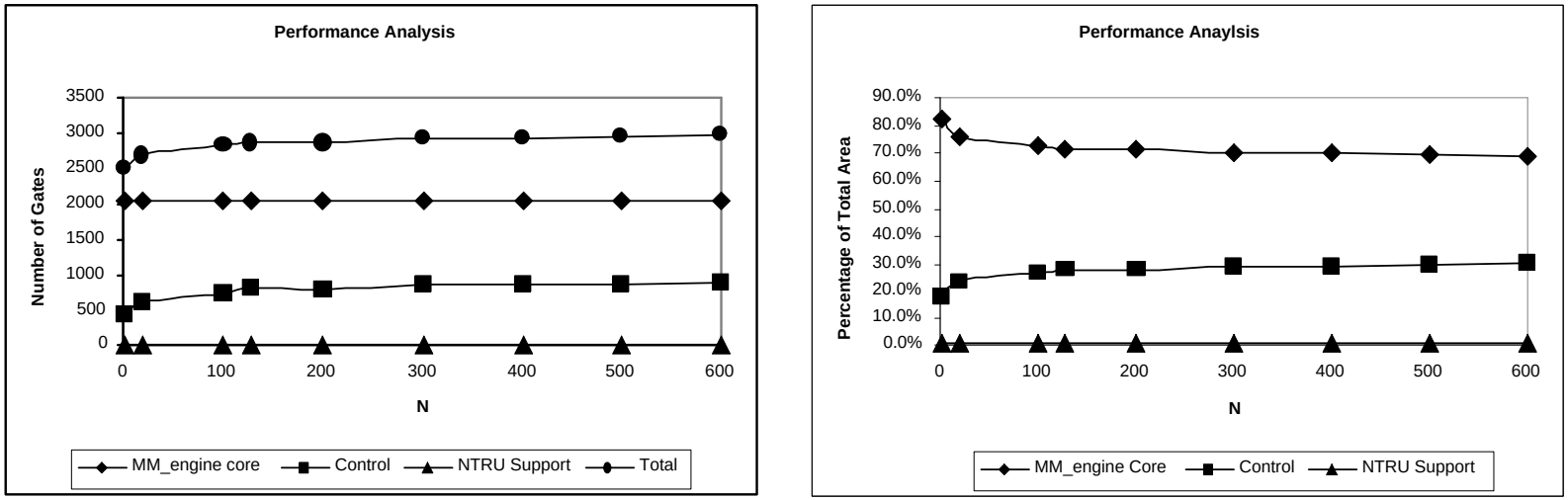


Figure 14.1: Gate Count (a) and Percentage of Total Area (b) for several operand lengths

with a short exponent in about 4 ms. Whereas, the 1024-bit RSA operation with a long exponent would approximately take 382 ms to complete. For the $GF(2^k)$ case, the unified multiplier required approximately 15ms to complete a 160-bit Elliptic Curve operation. Finally, for NTRU's highest security case ($N = 503$), the unified multiplier would compute a polynomial multiplication in a little over 6 ms.

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

Table 14.2: Estimated Performance of Unified Design

Chapter 15

Conclusions

This thesis aimed to meet the following objectives. First, we sought to efficiently enhance the performance of the polynomial multiplications that occur within NTRU's procedures. In addition, we wanted to create a design that was scalable as well as optimized for specific integer moduli. The work, also, aimed to create another design that was flexible and compatible with various cryptosystems and applications. We feel we have meet these goals as outlined below.

The software implementations of this thesis provided insight on the functionality of NTRU, estimations of how fast NTRU can be executed on two different platforms, and the practicality of applying CRT to the convolution algorithm. After thorough research, it was discovered that CRT provides no improvement to the convolution algorithm because of the small coefficients of one of the operands. However, the benefits of applying CRT to the computation of the inverse polynomials can be pursued

in future research.

The scalable NTRU multiplier focused on optimizing NTRU's core and most time consuming operation, the polynomial multiplication. The architecture was designed to be scalable to provide high performance by exploiting the parallelism within the polynomial multiplication. The research has demonstrated that the NTRU multiplier presents a wide range of time/area configurations. For instance, for embedded applications where low power consumption is critical, a single processing unit design can perform a polynomial multiplication in 0.9 ms with less than 1500 gates. This is reasonable performance for a low powered device. Finally, this work realized that the current security standards might be obsolete with the future rise of higher computing power. Therefore, this design exploits arbitrary key lengths to support a wide range of security levels to keep up with the ever-changing security standards.

For the unified design, this research has demonstrated that the Montgomery Multiplication algorithm can be used to perform modular multiplications for $GF(p)$ and $GF(2^k)$ and polynomial multiplications for NTRU. In addition, only 10 gates of additional hardware are required for the Montgomery Multiplier core to provide support for NTRU. The performance and application analysis demonstrated that all three types of applications RSA, ECC, and NTRU can be executed by the unified architecture with high performance. It is interesting to note that 503 NTRU offers the best performance for its security level compared to the other two applications. With an additional 2 ms over a short exponent 1024-bit RSA operation, 503 NTRU provides

a security level comparable to 4096-bit RSA. Despite the differences in performance between these applications, this unified design is now capable of supporting a majority of public-key cryptosystems, such as NTRU [3], RSA [4], Diffie-Hellman [27], Elliptic Curves [5, 6], etc., to meet the needs of virtually any public-key operation.

Bibliography

[1] J. H. Silverman, "High-Speed Multiplication of Truncated Polynomials," Tech. Rep. 10, NTRU Cryptosystems, Inc., January 1999. Version 1.

[2] D. Bailey, D. Coffin, A. Elbrit, J. Silverman, and A. Woodbury, "NTRU in Constrained Devices," in *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2001* (Ç. Koç, D. Naccache, and C. Paar, eds.), (Paris, France), pp. 266–277, Springer-Verlag, May 2001.

[3] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A Ring Based Public Key Cryptosystem," in *Algorithmic Number Theory: Third International Symposium (ANTS 3)* (J. P. Buhler, ed.), vol. LNCS 1423, pp. 267–288, Springer-Verlag, June 21–25 1998.

[4] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, February 1978.

[5] N. Koblitz, "Elliptic Curve Cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203–209, 1987.

[6] A. J. Menezes, *Elliptic Curve Public Key Cryptosystems*. Boston, Massachusetts, USA: Kluwer Academic Publishers, 1993.

[7] J. Hoffstein and J. H. Silverman, "Optimizations for NTRU," in *Proceedings of Public Key Cryptography and Computational Number Theory*, de Gruyter, Warsaw, September 2000.

[8] J. H. Silverman, "Communitive NTRU: Pseudo-code Implementation," Tech. Rep. 1, NTRU Cryptosystems, Inc., August 1997. Version 2.

[9] H. Cohen, *A Course in Computational Algebraic Number Theory*. Berlin, Germany: Springer-Verlag, 1993.

[10] A. Karatsuba and Y. Ofman, "Multiplication of Multidigit Numbers on Automata," *Sov. Phys. Dokl. (English translation)*, vol. 7, no. 7, pp. 595–596, 1963.

[11] J. Hoffstein and J. Silverman, "Small Hamming Weight Products in Cryptography." preprint, September 2000.

[12] J. H. Silverman, "Almost Inverses and Fast NTRU Key Creation," Tech. Rep. 14, NTRU Cryptosystems, Inc., March 1999. Version 1.

[13] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, Florida, USA: CRC Press, 1997.

[14] Xilinx Inc., San Jose, California, USA, *Virtex^{TM} 2.5V Field Programmable Gate Arrays*, 1998.

[15] M. Flynn and S. Oberman, *Advanced Computer Arithmetic Design*. New York: John Wiley & Sons, INC., 2001.

[16] I. Koren, *Computer Arithmetic Algorithms*. Prentice-Hall, 1993.

[17] R. Katz, *Contemporary Logic Design*, ch. 5. Addison-Wesley Publishing Co., 1993.

[18] H. Ling, "High Speed Binary Adder," in *IBM Journal of Research and Development*, vol. 25 of *2-3*, pp. 156–166, May 1981.

[19] M. Graphics, "ADK HTML Data Book TSMC 0.35 Micron FAST." http://ge.ee.wustl.edu/HEP/ADK/HTMLdatabook/TSMC035databook.htm.

[20] E. Savaş, A. F. Tenca, and K. Koç, "A Scalable and Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^m)$," in *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2000* (Çetin K. Koç and C. Paar, eds.), (Berlin, Germany), pp. 277–292, Springer-Verlag, LNCS 1965 2000.

[21] J. Großschädl, "A Bit-Serial Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^m)$," in *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2001* (Ç. Koç, D. Naccache, and C. Paar, eds.), (Paris, France), pp. 206–223, Springer-Verlag, May 2001.

[22] J. Goodman and A. P. Chandrakasan, "An Energy-Efficient Reconfigurable Public-Key Cryptography Processor," *IEEE Journal of Solid-State Circuits*, vol. 36, pp. 1808–1820, November 2001.

[23] G. Gaubatz, "Versatile Montgomery Multiplier Architectures," Master's thesis, ECE Department, Worcester Polytechnic Institute, Worcester, Massachusetts, USA, May 2002. Work in Progress.

[24] P. L. Montgomery, "Modular Multiplication without Trial Division," *Mathematics of Computation*, vol. 44, pp. 519–521, April 1985.

[25] Ç. K. Koç, T. Acar, and B. Kaliski, "Analyzing and Comparing Montgomery Multiplication Algorithms," *IEEE Micro*, pp. 26–33, June 1996.

[26] Ç. K. Koç and T. Acar, "Montgomery Multplication in $GF(2^k)$," *Design, Codes, and Cryptography*, vol. 14, no. 1, pp. 57–69, 1998.

[27] W. Diffie and M. E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. IT-22, pp. 644–654, 1976.