Parameter Space for the Architecture of FFT-Based Montgomery Modular Multiplication

Donald Donglong Chen, *Student Member, IEEE*, Gavin Xiaoxu Yao, Ray C.C. Cheung, *Member, IEEE*, Derek Pao, *Member, IEEE*, and Çetin Kaya Koç, *Fellow, IEEE*

Abstract—Modular multiplication is the core operation in public-key cryptographic algorithms such as RSA and the Diffie-Hellman algorithm. The efficiency of the modular multiplier plays a crucial role in the performance of these cryptographic methods. In this paper, improvements to FFT-based Montgomery Modular Multiplication (FFTM^3) using carry-save arithmetic and pre-computation techniques are presented. Moreover, pseudo-Fermat number transform is used to enrich the supported operand sizes for the FFTM^3. The asymptotic complexity of our method is $O(l \log l \log \log l)$, which is the same as the Schönhage-Strassen multiplication algorithm (SSA). A systematic procedure to select suitable parameter set for the FFTM^3 is provided. Prototypes of the improved FFTM^3 multiplier with appropriate parameter sets are implemented on Xilinx Virtex-6 FPGA. Our method can perform 3,100-bit and 4,124-bit modular multiplications in 6.74 and 7.78 μs, respectively. It offers better computation latency and area-latency product compared to the state-of-the-art methods for operand size of 3,072-bit and above.

Index Terms—Schönhage-Strassen algorithm, number theoretic transform (NTT), Montgomery modular multiplication, parallel computation, field-programmable gate array (FPGA)

1 Introduction

The multiplication of large integers is one of the core operations in public key cryptography. In order to provide the required cryptographic strength, the operand size has been growing continuously. During the earlier days of the RSA algorithm [1], the researchers believed that 512-bit size was sufficient; a few years of research in factoring RSA moduli immediately brought the key size to 1,024 bits. Currently many implementations already increase their key size to 2,048 bit, for example, the root keys issued by Verisign for SSL, while NIST [2] recommends 3,072-bit keys for protection beyond the year 2030.[1] Therefore, using RSA to provide long term protection, 3,072-bit or even larger integer modular multiplications need to be performed. Consequently, high-performance long integer modular multiplier is in demand for practical use of the RSA.

1.1 State of the Arts

Montgomery modular multiplication (MMM) [3] algorithm is the most popular algorithm to perform modular multiplications to date. It has been extensively studied, and several variants of MMM have been proposed for both hardware and software platforms [4], [5], [6], [7], [8], [9], [10], [11]. To compute $xy \bmod n$ using the original MMM, three l-bit multiplications dominate the computation time (l is the bitlength of n) [3]. This indicates the acceleration of multiplication will benefit the performance of MMM significantly.

The asymptotic complexities of multiplication algorithms from the schoolbook method to the Fürer method are listed in Table 1. The GMP library [16] provides efficient software implementations for most of these algorithms. However, it only focuses on software. There are many hardware realizations of these multiplication algorithms: the schoolbook [4], [5], [9], [10] and Karatsuba methods [12], [17], [18], but few on Schönhage-Strassen algorithm (SSA) [7].

Saldamlı and Koç [19] proposed an algorithm to perform the whole MMM in spectral domain. However, their spectral modular algorithm is derived from the digit-serial variant of MMM [20], and as its name indicated, such algorithm is essentially sequential, thus not friendly to massively parallel computation.

McLaughlin [21] proposed a new framework for a modified version of MMM which the multiplications are suitable to perform in spectral (frequency) domain. The new version of MMM have lower multiplication time than the original version MMM. Moreover, by using cyclic convolution and negacyclic convolution to compute the multiplication, the framework could avoid doubling the length of transforms.

The FFT-based Montgomery product reduction (FMPR) algorithm, which performs only the multiplication of MMM in spectral domain was proposed by David et al. [7]. The FFT/IFFT and component-wise multiplication in FMPR algorithm are suitable for a parallel hardware design.

1. This estimation is based on the general number field sieve attack, the fastest known attack on RSA. If there are significant breakthroughs in cryptanalysis, one may need either longer keys or change of algorithms.

- *D. Chen, G. Yao, R. Cheung, and D. Pao are with the Department of Electronic Engineering, City University of Hong Kong, Hong Kong. E-mail: {donald.chen, gavin.yao}@my.cityu.edu.hk, {r.cheung, d.pao}@cityu.edu.hk.*
- *Ç.K. Koç is with the Department of Computer Science, University of California Santa Barbara, Santa Barbara, CA. E-mail: koc@cs.ucsb.edu.*

Manuscript received 5 Aug. 2014; revised 30 Jan. 2015; accepted 16 Feb. 2015. Date of publication 26 Mar. 2015; date of current version 16 Dec. 2015. Recommended for acceptance by P. Tang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2015.2417553

TABLE 1
Multiplication Algorithms and Their Bit-Level Complexity

Algorithm	Complexity
Schoolbook	$O(l^2)$
Karatsuba [12]	$O(l^{\log 3/\log 2}) \approx O(l^{1.585})$
3-way Toom-Cook [13]	$O(l^{\log 5/\log 3}) \approx O(l^{1.465})$
k-way Toom-Cook [13]	$O(l^{\log(2k-1)/\log k})^{\star}$
Schönhage-Strassen [14]	$O(l \cdot \log l \cdot \log\log l)$
Fürer [15]	$O(l \cdot \log l \cdot 2^{O(\log^* l)})^{\dagger}$

* *k represents the number of parts the operands are divided to.*$^{\dagger}\log^* l$ *represents the iterated logarithm operation.*

1.2 Our Contributions

The motivation of this paper is to provide a systematic parameter selection method for the FFT-based MMM, explore the arithmetic as well as the hardware improvements for it, and compare the performance when the FFT technique and the other multiplication algorithms are used in MMM.

Notice that there is still no hardware realizations on McLaughlin's improved MMM, in order to have a fair comparison and focus on the performance gain by using the FFT technique versus other multiplication methods, we target on the improvements and implementation for the FMPR algorithm in this paper and left the McLaughlin's algorithm algorithm as the future work.

There are several places in the FMPR algorithm that can be improved to achieve a better performance and a higher computation resources utilization. First, in FMPR, long accumulation is required in IFFT, which is the bottleneck for a parallel hardware design. Second, the long carry-chain in the time domain addition is not suitable for a high frequency hardware design. Most importantly, the gap of the supported operand size between two FMPRs becomes larger and larger when the FFT length increased. This implies that one may have to use the parameter set which has much larger operand size than the actual need. This will cause a waste of computation resources.

Improved from the FMPR algorithm [7], we interleave the multiplication and addition in spectral domain, while performing the reduction and division in time domain during the computation of Montgomery modular multiplication. We rename it as FFT-based Montgomery modular multiplication (FFTM^3) to distinguish the differences. The main contributions of this paper are as follows:

- Non-least positive (NLP) form and carry-save technique are used in our work to translate the long accumulation into short carry-save addition, which make a parallel accumulation design feasible;
- Pseudo-Fermat number transform [22] is exploited to enrich the supported operand sizes of FFTM^3;
- In order to facilitate the usage of FFTM^3 to different applications. The parameter specifications for FFTM^3 are analyzed; A systematic parameter selection method is proposed for the efficient selection of parameters for a targeted operand size modular multiplication;
- Pipelined architectures are designed for FFTM^3 and the implementation results from 1,024-bit to 15,484-bit are provided. To compute one 4,096-bit modular multiplication, our FFTM^3 could have a 2.43 times speedup when compared with the state of the art Montgomery multiplier in [10].

The rest of this paper is organized as follows. Section 2 recaps the mathematical backgrounds. In Section 3, the FFTM^3 algorithm is revisited. The parameter set specifications are analyzed in Section 4. Section 5 provides further improvements for the FFTM^3 algorithm. Section 6 proposes the efficient parameter sets selection method for a targeted operand size FFTM^3. In Section 7, the hardware architecture design is described in detail. Section 8 provides the FPGA implementation results and the comparison with other works. Section 9 concludes this paper.

TABLE 2
Notation List

Notation	Definition	Remark
$x(t)$	Polynomial representation of x in time domain	
$X(k)$	Representation of $x(t)$ in spectral domain	
Notation	**Definition**	**Remark**
b	Radix of the representation	
μ	Bitlength of the word (word size)	$b = 2^{\mu}$
q	Ring size for NTT	$q = 2^{v} + 1$
v	The constant to construct q	
τ, δ	Numbers to construct v	$v = 2^{\tau}$ or $v = \delta 2^{\tau}$
d	Length of the NTT	$d = 2s$
ω	Primitive d-th root of unity in $\mathbb{Z}_q$	$\omega^d = 1 \bmod q$
n	Modulus of the MMM	n is odd
l	Bitlength of n	
r	A power of 2 greater than $4n$	$r = b^s = 2^{\mu s}$
s	Number of radix-b words in n	$l \leq s\mu$

2 Backgrounds

This section provides the mathematical background about the Schönhage-Strassen algorithm and the Montgomery modular multiplication. For the ease of reference, the symbols in this paper with their definitions are summarized in Table 2.

2.1 Number Theoretic Transform (NTT)

Let an integer $x = \sum_{i=0}^{s-1} x_i b^i := (x_{s-1}, \ldots, x_1, x_0)_b$ be the representation in the radix-b positional number system. Then x can be also represented as a $(s-1)$-degree polynomial: $x(t) = x_{s-1}t^{s-1} + \cdots x_1 t + x_0$ and therefore, $x = x(b)$. In computer arithmetic, b is usually a power of 2, i.e., $b = 2^{\mu}$. These μ-bit coefficients are also known as *words*.

For instance, $x = 155 = (2123)_4$ can be represented by a radix-4 polynomial as:

$$x(t) = 2t^3 + 1t^2 + 2t + 3. \tag{1}$$

In polynomial representations, the multiplication is the *linear convolution* of the coefficients. Computing $z = x \cdot y$ is equivalent to:

$$z_i = \text{linear\_conv}(x(t), y(t))_i := \sum_{j=0}^{s-1} x_{(i-j)} \cdot y_j \tag{2}$$

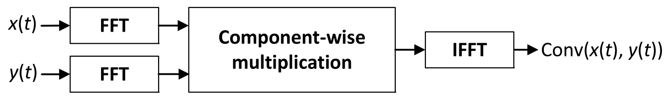


Fig. 1. Data flow of Schönhage-Strassen algorithm.

for all $0 \leq i < 2s-1$. The coefficient will be set to 0 if its index j or $(i-j)$ is out of $[0, s)$.

Similar to the discrete Fourier transform (DFT), the number theoretic transform provides a special domain where the component-wise multiplication is equivalent to the convolution in the normal representation [23]. Inherited from DFT, we call the transformed domain as *spectral domain* and the normal space as *time domain*.

NTT is defined over a finite ring $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ [23]. Let $X(k)$ be the polynomial in spectral domain, and x_i, X_i be the ith coefficients of $x(t)$ and $X(k)$. Let ω be a primitive dth root of unity in $\mathbb{Z}_q$. Let $x(t)$ and $X(k)$ be polynomials of degree less than d, where $d \in \mathbb{N}$. The length-d NTT and its inverse (INTT) are defined as:

$$X_i = \mathrm{NTT}(x(t))_i := \sum_{j=0}^{d-1} x_j \omega^{ij} \mod q \tag{3}$$

$$x_i = \mathrm{INTT}(X(k))_i := d^{-1} \sum_{j=0}^{d-1} X_j \omega^{-ij} \mod q, \tag{4}$$

where $i = 0, 1, \ldots, d-1$, and $d \cdot d^{-1} \equiv 1 \bmod q$.

Pollard [23] proved that the fast Fourier transform (FFT) is also available to NTT. To compute all the coefficients, (3) or (4) is with word-level complexity of $O(d^2)$, while FFT $O(d \log d)$[24]. The jth stage decimation-in-time equation for an FFT is as follows $(0 \leq j \leq \log_2 d - 1)$:

$$\begin{cases} X_k = x_{2k} + x_{2k+1}\omega^{P_{jk}} \mod q \\ X_{k+\frac{d}{2}} = x_{2k} - x_{2k+1}\omega^{P_{jk}} \mod q, \end{cases} \tag{5}$$

where $P_{jk} = \lfloor \frac{k}{2^j} \rfloor \times 2^j$, $k = 0, 1, \ldots, \frac{d}{2} - 1$. The operations are denoted as $\mathrm{FFT}(x(t))$ and $\mathrm{IFFT}(X(k))$.

Let $X(k) = \mathrm{FFT}(x(t)), Y(k) = \mathrm{FFT}(y(t))$ and $\odot$ be component-wise multiplication. Then the *cyclic convolution* can be computed as follows:

$$\begin{aligned} \mathrm{cyclic\_conv}(x(t), y(t))_i &:= \sum_{j=0}^{d-1} x_{(i-j) \bmod d} \cdot y_j \\ &= \mathrm{IFFT}(X(k) \odot Y(k))_i. \end{aligned} \tag{6}$$

Note that the component-wise multiplication $X(k) \odot Y(k)$ is with the word-level complexity of $O(d)$.

Not every length-d NTT in $\mathbb{Z}_q$ supports cyclic convolutions. Nussbaumer [25] proved that the cyclic convolution is available if and only if d divides $p-1$ for every prime factor p of q.

2.2 Schönhage-Strassen Algorithm

The basic idea of SSA is to perform the multiplication in spectral domain by using FFT and IFFT for transforms. The data flow of SSA is shown in Fig. 1. Since the word-level complexity of FFT and IFFT is $O(d \log d)$ and it is $O(d)$ for multiplication, the entire complexity is sub-quadratic.

In order to accelerate the operations in SSA, the parameters to construct the ring $\mathbb{Z}_q$ are of special form:

- q is a Fermat number of form $2^v + 1$, where v is a power of 2
- ω is a power of 2
- d is a power of 2.

Since q is in the form $2^v + 1$, which $2^v \equiv -1 \bmod q$, fast modular reduction arithmetic is available. Let $(x_{\beta-1}x_{\beta-2}\ldots x_1x_0)_{2^v}$ be the radix-2^v representation of x. Then

$$x \equiv \sum_{i=0}^{\beta-1} x_i \cdot (-1)^i \mod q. \tag{7}$$

The choice of ω ensures that a multiplication by the power of ω can be simply achieved by shift operation. The selection of d as a power of 2 makes the radix-2 FFT structure available.

The selection of q, ω and d above also makes sure that d^{-1} is also a power of 2; Since $\omega^d = 2^{d \log_2 \omega} = 1 \bmod q$, then,

$$d^{-1} = 2^{-\log_2 d} = 2^{d \log_2 \omega} 2^{-\log_2 d} = 2^{d \log_2 \omega - \log_2 d} \mod q.$$

Since both ω and d are powers of 2, d^{-1} is a power of 2.

Note that the linear convolution (2) is slightly different to the cyclic convolution (6). In order to make the cyclic convolution equivalent to the linear convolution, s should satisfy $s \leq d/2$. Also, to avoid overflow, the size of ring $\mathbb{Z}_q$ should be greater than the largest coefficient of the product, i.e. $q > sb^2$. Therefore, when $s = d/2$ and b is the largest power of 2 that satisfies $sb^2 < q$, ring $\mathbb{Z}_q$ provides the largest dynamic range for SSA.

2.3 Montgomery Modular Multiplication

Montgomery modular multiplication [3] is an efficient modular algorithm when the modulus n is without specific form. By adding constraints on the parameters, Walter [26] proposed the MMM without conditional subtraction algorithm and it is given in Algorithm 1. In MMM, r is typically a power of 2 for the ease of computation.

Algorithm 1. Montgomery Modular Multiplication Without Conditional Subtraction [26]

Copirme integers n *and* $r, r > 4n, x \equiv x'r \bmod n, x < 2n, y \equiv y'r \bmod n, y < 2n, n' \equiv -n^{-1} \bmod r$.

```
Input: $x, y, n, n'$, and $r$
Output: $z \equiv xyr^{-1} = x'y'r \bmod n, z < 2n$
1: $g \leftarrow x \cdot y$
2: $m \leftarrow (g \bmod r)n' \bmod r$
3: $z \leftarrow (g + mn)/r$
4: return $z$
```

3 FFT-Based MMM Algorithm

The FFT-based Montgomery product reduction algorithm in [7] is based on the original MMM. In order to avoid the conditional subtraction, SSA is applied to the MMM without conditional subtraction as shown in Algorithm 1.

Furthermore, as shown in Algorithm 2, the addition in MMM is also performed in spectral domain and the benefits are three-fold.

- The l-bit time domain addition is transformed to a number of independent $(v+2)$-bit additions in spectral domain. The l-bit long carry chain is divided into $(v+2)$-bit ones.
- Compared with the original algorithm, no further transform (FFT/IFFT) is required.
- The $(v+2)$-bit addition is performed following the component-wise multiplication, which is more suitable for a pipelined architecture design.

Algorithm 2. FFT-Based Montgomery Modular Multiplication

x, y, n, and n' are as required by Algorithm 1. Suppose that there exists a length-d NTT for a primitive root of unity ω in $\mathbb{Z}_q$. Let $s = d/2$, $r = b^s > 4n$. Let $x(t)$, $y(t)$, $n(t)$, and $n'(t)$ be the time domain polynomial of x, y, n, and n', which satisfy $x(b) = x$, $y(b) = y$, $n(b) = n$, and $n'(b) = n'$. $X(k)$, $Y(k)$, $N(k)$, and $N'(k)$ are the spectral domain polynomials of $x(t)$, $y(t)$, $n(t)$, and $n'(t)$, respectively.

```
Input: $X(k), Y(k), N(k), N'(k)$, $q$ and $r$
Output: $Z(k) = \mathrm{FFT}(z(t))$ where $z = xyr^{-1} \bmod n$
 1: $G(k) \leftarrow X(k) \odot Y(k) \bmod q$
 2: $g \leftarrow \mathrm{IFFT}(G(k))$
 3: $h \leftarrow g \bmod r$
 4: $H(k) \leftarrow \mathrm{FFT}(h)$
 5: $M(k) \leftarrow H(k) \odot N'(k) \bmod q$
 6: $m \leftarrow \mathrm{IFFT}(M(k))$
 7: $m \leftarrow m \bmod r$
 8: $M(k) \leftarrow \mathrm{FFT}(m)$
 9: $K(k) \leftarrow M(k) \odot N(k) \bmod q$
10: $Z(k) \leftarrow K(k) + G(k) \bmod q$
11: $z \leftarrow \mathrm{IFFT}(K(k))$
12: $z \leftarrow z/r$
13: $Z(k) \leftarrow \mathrm{FFT}(z)$
14: return $Z(k)$
```

The FFTM^3 is less complex than three separated SSA multiplications. Specifically, there are six FFT/IFFT in FFTM^3 instead of nine in three SSA multiplications. This is because in modular exponentiation, $N'(k)$ and $N(k)$ are precomputed and can be reused. Also, by using the MSB-first square-and-multiply algorithm in exponentiation, the computation is either square or multiplication by a fixed number, thus, only one input needs FFT.

4 Parameter Specifications and Flexible Operand Size FFTM^3

4.1 Parameter Specifications for FFTM^3

By carefully selecting $r = b^s = 2^{\mu s}$, modulo-r and division-by-r operations can be simply performed by words selection. The MMM algorithm without conditional subtraction requires that $4n < r$, therefore, the operand size of n should satisfies $l \le \mu s - 2$.

Furthermore, according to (2), the maximum value of the coefficient after convolution (component-wise multiplication in spectral domain) is sb^2. The addition in Step 10 doubles the size of the coefficients, thus, to avoid overflow, the parameters need to satisfy $s \le d/2$ and

$$q > 2sb^2 = s \cdot 2^{2\mu+1}.$$

In order to utilize the regular radix-2 FFT/IFFT, one choice for q are Fermat numbers in the form $2^v + 1$ which $v = 2^\tau$. It is proved in [25] that when $2^{2^\tau} + 1$ is composite ($\tau \ge 5$), one can always define NTT modulo $q = 2^{2^\tau} + 1$ of length $d = 2^{\tau+1-i}$ with root $\omega = 2^{2^i}$ where $i \in \mathbb{Z}$. This deduction indicates that for a fixed Fermat number, a smaller root ω is accompanied by a longer length d, which enables a larger operand size.

The parameter specifications for FFTM^3 are summarized as follows:

1) q is number of form $2^{2^\tau} + 1$.
2) $d = 2^{\tau+1-i}$ with $\omega = 2^{2^i}$ for integer i.
3) The greatest $b = 2^\mu$ such that $2sb^2 = s \cdot 2^{2\mu+1} < q$ where $s = d/2$.
4) $l = \mu s - 2$, where l is the maximum operand size of n.

The sequence of the specifications also reflects the parameter selection procedure, and it starts from the selection of τ. Following the above parameter specifications, we carefully select six parameter sets for FFTM^3 with operand size range from 926-bit to 7,678-bit, and it is shown in Table 3.

TABLE 3
Parameter Set Selection for FFTM^3

# Bits l	Ring $\mathbb{Z}_q$	NTT length d	Root ω	Word size μ	# Words s
926	$2^{64}+1$	64	4	29	32
1,790	$2^{64}+1$	128	2	28	64
3,838	$2^{128}+1$	128	4	60	64
7,678	$2^{128}+1$	256	2	60	128

4.2 Flexible Operand Size FFTM^3

As shown in Table 3, the operand sizes which can be chosen for modular multiplication cover the main key length requirement nowadays [2], [27]. However, the gap between each operand sizes become larger and larger when the values of q and d increase.

This implies that one may have to use the parameter set which has a much larger operand size than the actual needed one. For example, for key size 2,048-bit which is recommended for RSA by NIST, one has to perform modular multiplication by using the 3,838-bit parameter set (which is 1,790-bit larger than the required operand size) from Table 3. Therefore, the construction of parameter sets whose operand sizes are closed to the target size is preferred.

As discussed in the previous section, given a fixed d, ring $\mathbb{Z}_q$ can provide the largest dynamic range if the largest μ which satisfied $q > 2^{2\mu+1}s$ is selected. This indicates if the value of q is more flexible when given a fixed s, the operand size l will be more various. In order to meet this requirement, we introduce the pseudo-Fermat number q which is of the form $2^{\delta 2^\tau} + 1$ [22]. The Fermat or pseudo-Fermat number q and its relationship with root and NTT length are listed in Table 4.

TABLE 4
Relationship between Fermat or Pseudo-Fermat Ring q, NTT length d, and the Root ω

Ring $\mathbb{Z}_q$	NTT length d	Root ω
$2^{\delta 2^\tau}+1, \delta \geq 2,$ δ power of 2, $\tau \geq 2$	$2^{\tau+1}$	2^δ
$2^{2^\tau}+1, \tau > 0$	$2^{\tau+1}$	2
$2^{2^\tau}+1, \tau \geq 2$	$2^{\tau+2}$	$2^{2^{\tau-2}}(2^{2^{\tau-1}}-1)$
$2^{\delta 2^\tau}+1, \delta \geq 2,$ δ not power of 2, $\tau \geq 2$	$2^{\tau+1}$	2^δ

After the introduction of pseudo-Fermat number transform, the number of parameter sets one can obtain for a target operand size is increased. It can be observed that the introduction of variable δ enriches the value of q, which makes the selection of parameter sets more flexible.

5 Further Improvements for FFTM^3

The transform from normal positional system to polynomial representation is trivial; however, the reverse transform needs long accumulation. A pipelined addition design [28] could speedup the accumulation process, however, the computation is still in serial. In this section, the non-least-positive form and the carry-save arithmetic (CSA) are introduced to parallelize this long serial accumulation. Moreover, their effects to the FFTM^3 algorithm are also discussed.

5.1 NLP Form and Carry-Save Arithmetics

In Section 2, all the words in the positional system or the coefficients in the polynomial satisfy that $0 \leq x_i < b$. In this case the representation is in its *least-positive (LP) form*. Note that for a given value, its LP form representation is unique.

However, the words or the coefficients do not necessarily fall into the range of $[0, b)$. When there are words or coefficients out of this range, the representation is in its *NLP form*. Note that such NLP representation is not unique. Taking the same example, $x = (2123)_4$ can be written in an NLP polynomial and it represents the same value as (1):

$$x_{\text{NLP}}(t) = 3t^3 - 5t^2 + 10t + 3. \quad (8)$$

The NLP form representation can be transformed to LP form by carry propagation, where the quotients $\lfloor x_i/b \rfloor$ are considered as carries. The direction of propagation is from the lowest degree word to the highest degree one.

If NLP form is tolerable in the FFTM^3 computation, then one could use CSA to control the carry propagation length in the large size addition. In long integer operations, the carry should be passed from the least significant bit all through to the most significant bit. This carry propagation chain is usually the longest data path, i.e., the bottleneck of the performance improvement. Using the CSA can minimize the propagation length effectively, and hence, improve the performance.

Specifically, we need to accumulate the coefficients and generate the normal positional integer after IFFT. We still use italic $z(t)$ and z_i to denote the polynomial in time domain and its ith coefficient after IFFT, while using

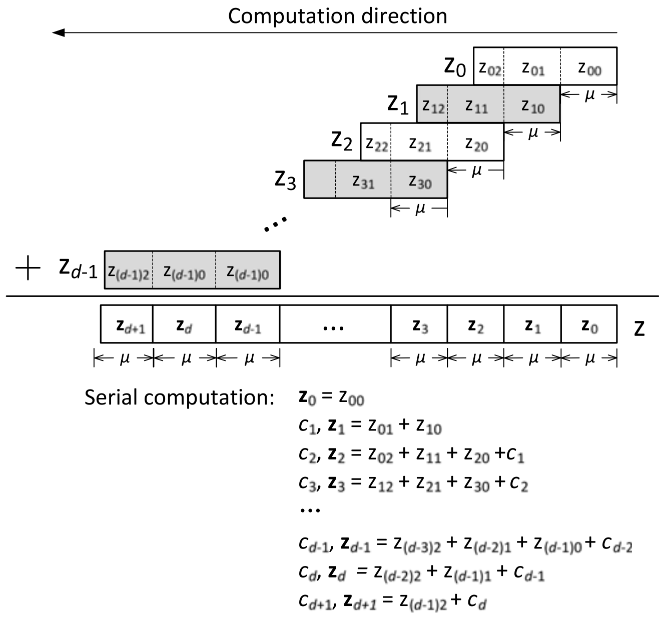

Computation direction
z0 z02 z01 z00
z1 z12 z11 z10
z2 z22 z21 z20
z3 z31 z30
μ
+ zd-1 z(d-1)2 z(d-1)0 z(d-1)0
zd+1 zd zd-1 ... z3 z2 z1 z0 z
Serial computation: z0 = z00
c1, z1 = z01 + z10
c2, z2 = z02 + z11 + z20 + c1
c3, z3 = z12 + z21 + z30 + c2
...
cd-1, zd-1 = z(d-3)2 + z(d-2)1 + z(d-1)0 + cd-2
cd, zd = z(d-2)2 + z(d-1)1 + cd-1
cd+1, zd+1 = z(d-1)2 + cd

Fig. 2. Computation of the time domain coefficients using serial adders. Bold type $\mathbf{z}_i$ is used to represent the output coefficient.

boldface $\mathbf{z}$ and $\mathbf{z}_i$ to denote the normal positional integer and its ith word after the accumulation. That is:

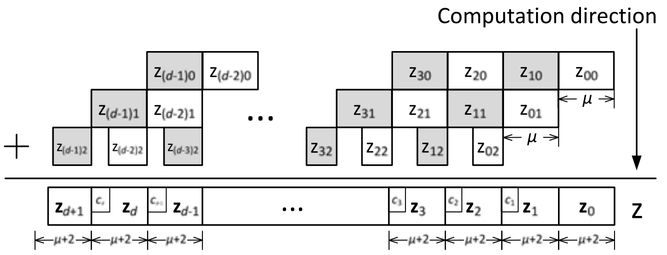
$$\mathbf{z} = z(b) = z_{d-1}b^{d-1} + \cdots + z_1 b + z_0. \quad (9)$$

Let B be the number of segments which a coefficient is divided. Thus the bit-width of coefficient and B satisfy $v + 2 < B\mu$ $(B \geq 2)$. One can chop z_i into $B-1$ μ-bit and one $(v + 2 - (B-1)\mu)$-bit *segments*. We take $B = 3$ as an example here (the most frequently appear case) and chop z_i into z_{i0}, z_{i1}, z_{i2}. The accumulation of (9) can be performed as Fig. 2 to get the LP form representation of $\mathbf{z}$. As the carries require to be propagated from z_{00} to $z_{(d-1)2}$, the data dependency limits the computation in serial.

On the other hand, if NLP form polynomial is tolerable in the FFTM^3, we can use carry-save adders to minimize the long carry propagation. Fig. 3 depicts the parallel carry-save addition for the computation of the time domain NLP coefficients. Since the largest value of $\mathbf{z}_i$ is the sum of two μ-bit segments z_{i0}, $z_{(i-1)1}$ and one $(v + 2 - 2\mu)$-bit segment $z_{(i-2)2}$, the carry value is at most 2. Thus, only two more bits are needed to store the carry c_i for each word. In this way, the additions are independent, and can be performed in parallel. Compared with the serial accumulation process, the carry-save accumulation[2] can achieve a speedup by a factor up to s.

Note that carry look ahead framework (CLA) can be used after CSA to transform the NLP form result to LP form. However, more hardware resources are required to build this framework, especially in our case where the word size μ is usually not small. As it will be shown in the later subsection that using NLP form only has limited affects on FFTM^3, CLA is not applied in our design for a resource saving design.

2. The CSA proposed here leaves the output in carry-stored representation


Computation direction
$z_{(d-1)0}$ $z_{(d-2)0}$
$z_{(d-1)1}$ $z_{(d-2)1}$
$z_{(d-1)2}$ $z_{(d-2)2}$ $z_{(d-3)2}$
...
z_{30} z_{20} z_{10} z_{00}
z_{31} z_{21} z_{11} z_{01}
z_{32} z_{22} z_{12} z_{02}
μ
+
$\mathbf{z}_{d+1}$ c_d $\mathbf{z}_d$ c_{d-1} $\mathbf{z}_{d-1}$ ... c_3 $\mathbf{z}_3$ c_2 $\mathbf{z}_2$ c_1 $\mathbf{z}_1$ $\mathbf{z}_0$
Z
$\mu+2$

Parallel computation: $\mathbf{z}_0 = z_{00}$

$c_1, \mathbf{z}_1 = z_{01} + z_{10}$

$c_2, \mathbf{z}_2 = z_{02} + z_{11} + z_{20}$

$c_3, \mathbf{z}_3 = z_{12} + z_{21} + z_{30}$

...

$c_{d-1}, \mathbf{z}_{d-1} = z_{(d-3)2} + z_{(d-2)1} + z_{(d-1)0}$

$c_d, \mathbf{z}_d = z_{(d-2)2} + z_{(d-1)1}$

$\mathbf{z}_{d+1} = z_{(d-1)2}$

Fig. 3. Computation of the time domain coefficients using parallel adders. Bold type $\mathbf{z}_i$ is used to represent the output coefficient.

5.2 Relieving the Computation Efforts for the Zero Coefficients

It is worth to note that the zero coefficients can be eliminated from the accumulation process to speedup the computation; in modulo-r operation, the higher s coefficients are not necessarily generated. Similarly, in division-by-r, only the higher $s+1$ coefficients are needed. Therefore, half of the accumulation time can be further saved, and Steps 3, 7 and 12 of Algorithm 2 can be computed during the IFFT operations.

5.3 Modulo-r Operation

The modulo-r results for LP form and NLP form may be different. Taking polynomials (1) and (8) as an example, let r be $b^2 = 16$, then the two modulo-r residues are:

$$z_{\mathrm{LP}} = x_{\mathrm{LP}}(b) \bmod r = 2b + 3 = 11,$$

$$z_{\mathrm{NLP}} = x_{\mathrm{NLP}}(b) \bmod r = 10b + 3 = 43.$$

Though the two results are congruent modulo-r, the residue of NLP form polynomial is still in NLP form.

In order to make the NLP form result close to the least-positive residue, the carry of $\mathbf{z}_{s-1}$ can be dropped, hence, the maximum value for $\mathbf{z}_{s-1}$ is $2^\mu - 1$. As the carry c_{s-2} from $\mathbf{z}_{s-2}$ is at most 2, the maximum value of z_{s-1} is $2^\mu + 1$ after carry propagation. Therefore, the value of NLP residue is at most $(2^\mu + 1)b^{s-1} + b^{s-1} - 1 < (2^\mu + 2) \cdot b^{s-1} < 2r$. This means the NLP residue equals to either the LP residue or the LP residue plus r.

Following the above derivation, if the result in Step 4 of Algorithm 2 is $h + r$ instead of h, m in Step 6 would be $(h+r)n'$ instead of hn'. However, this additional rn' will be eliminated by modulo-r operation in Step 7.

Similarly, if the result in Step 7 is $m + r$ instead of m, the value of z in Step 10 would be $t + (m+r)n = (t + mn) + rn$, hence, the result in Step 12 would be $(t + mn)/r + n = z + n$, which has a difference of n from the least-positive result. Nevertheless, since $z \equiv z + n \bmod n$, as long as there is enough dynamic range, the NLP result will not affect the following operations.

5.4 Division-by-r Operation

For the division-by-r operation of NLP form polynomial, error correction may be needed. Denote ϵ the result of the modulo-r NLP residue divided by r, then one can first select all words except the lowest s ones, and then add ϵ to get the corrected *quotient*.

Still take $r = b^2 = 16$ and polynomial (8) as example:

$$\epsilon = \lfloor (x_{\mathrm{NLP}}(b) \bmod r)/r \rfloor = \lfloor 43/16 \rfloor = 2,$$
$$\lfloor x/r \rfloor = x_{\mathrm{NLP}}(b)/r + \epsilon = 3b - 5 + 2 = 9.$$

Note that ϵ might not be equal to $\lfloor x_{s-1}/b \rfloor$ because the carry from lower words may affect this value. Therefore, compute ϵ is troublesome: one needs to first propagate the carries from the lowest word to the $(s-1)$th word, which is still a serial operation.

In order to speedup the computation, we developed a fast method to compute ϵ in FFTM^3 without serial accumulation. Note that the division-by-r operation only appears in Step 12 of Algorithm 2, and the input $\mathbf{z} = z(b)$ of Step 12, i.e., the output of Step 11, satisfies the following fact:

Fact 1. MMM algorithm guarantees that $(g + mn)$ is divisible by r, thus, if the polynomial is in LP form, coefficients $\mathbf{z}_{s-1}, \mathbf{z}_{s-2}, \ldots, \mathbf{z}_0$ of $\mathbf{z}$ should all be zeros in Step 11 of Algorithm 2.

Thanks to Fact 1, we can use the following computation to determine ϵ (still taking $B = 3$ as an example, the method for other B values are similar):

$$(c_{s-1}, \mathbf{z}_{s-1}) \leftarrow z_{(s-1)0} + z_{(s-2)1} + z_{(s-3)2}. \tag{10}$$

If the inputs of IFFT accumulation are all positive or zero, then ϵ is determined by the condition of $\mathbf{z}_{s-1}$:

- If $\mathbf{z}_{s-1} = 0$, the lower words must all be zero in order to satisfy Fact 1. Therefore, $\epsilon = c_{s-1}$.
- If $\mathbf{z}_{s-1} > 0$, the lower words must send a carry c' so that $\mathbf{z}_{s-1} + c' = b = 2^\mu$ in order to satisfy Fact 1. In this situation, $\epsilon = c_{s-1} + 1$

The above computation can be further simplified by only using the two most significant bits of segments $z_{(s-1)0}$ and $z_{(s-2)1}$. As long as $z_{(s-3)2}$ is within $(\mu - 2)$ bits, the carry-in from the lower bits of $\mathbf{z}_{s-1}$ could only be 0, 1, or 2, the two highest bits can already absorb this carry-in. Therefore,

$$\epsilon = \left\lceil \frac{\lfloor 4z_{(s-1)0}/b \rfloor + \lfloor 4z_{(s-2)1}/b \rfloor}{4} \right\rceil. \tag{11}$$

In hardware realization, using only a few gates one can produce the ϵ value.

5.5 Modulo-q Operation

Note that modulo-q reduction is in great demand in FFTM^3, hence a fast computation can speedup the whole system significantly. Again, NLP form can be utilized. Recall Equation (7), normally, one needs a correction step after computing $\sum x_i \cdot (-1)^i$ to ensure the result in the range $[0, 2^v]$. To enable fast computation, the result $\sum x_i \cdot (-1)^i$ is used directly without the correction step.

Although $\sum x_i \cdot (-1)^i$ is probably not in range $[0, 2^v + 1)$, this NLP value is tolerable for the operations in $\mathbb{Z}_q$. For

TABLE 5
Computational Requirement of FFTM^3 Using Fermat or Pseudo-Fermat Number Transform

Operation	Basic operation	Equivalent operation	Number of equivalent operation
Component-wise multiplication	$(v+2)$-bit MUL	-	d
	$(2v+2)$-bit MR	$\lfloor\frac{2v+2}{v}\rfloor$ v-bit A/S	$2d$
FFT, $\omega \neq \sqrt{2}$	$(v+2)$-bit A/S	$(v+2)$-bit A/S	$d\log_2 d$
	$(v+3)$-bit MR	$\lfloor\frac{v+3}{v}\rfloor$ v-bit A/S	$d\log_2 d$
	$(v+2+(\frac{d}{2}-1)\log_2\omega)$-bit SH & MR	$\lfloor\frac{(v+2+(\frac{d}{2}-1)\log_2\omega)}{v}\rfloor$ v-bit A/S	$\frac{d}{2}\log_2 d$
IFFT, $\omega \neq \sqrt{2}$	$(v+2)$-bit A/S	$(v+2)$-bit A/S	$d\log_2 d + d$
	$(v+3)$-bit MR	$\lfloor\frac{v+3}{v}\rfloor$ v-bit A/S	$d\log_2 d + d$
	$(v+2+(\frac{d}{2}-1)\log_2\omega)$-bit SH & MR	$\lfloor\frac{(v+2+(\frac{d}{2}-1)\log_2\omega)}{v}\rfloor$ v-bit A/S	$\frac{d}{2}\log_2 d$
	$\lceil\frac{v+2}{\mu}\rceil$-input μ-bit A/S	$\lfloor\frac{v+2}{\mu}\rfloor$ μ-bit A/S	$\frac{d}{2}+1$
FFT, $\omega = \sqrt{2}$	$(v+2)$-bit A/S	$(v+2)$-bit A/S	$d\log_2 d + \frac{d}{2}$
	$(v+3)$-bit MR	$\lfloor\frac{v+3}{v}\rfloor$ v-bit A/S	$d\log_2 d + \frac{d}{2}$
	$(v+2+\frac{d+3v}{4}-1)$-bit SH & MR	$\lfloor\frac{v+2+\frac{d+3v}{4}-1}{v}\rfloor$ v-bit A/S	$\frac{d}{2}$
	$(v+2+\frac{d+v}{4}-1)$-bit SH & MR	$\lfloor\frac{v+2+\frac{d+v}{4}-1}{v}\rfloor$ v-bit A/S	$\frac{d}{2}$
	$(v+2+\frac{d}{2}-2)$-bit SH & MR	$\lfloor\frac{(v+2+(\frac{d}{2}-1)\log_2\omega)}{v}\rfloor$ v-bit A/S	$\frac{d}{2}(\log_2 d - 1)$
IFFT, $\omega = \sqrt{2}$	$(v+2)$-bit A/S	$(v+2)$-bit A/S	$d\log_2 d + \frac{3d}{2}$
	$(v+3)$-bit MR	$\lfloor\frac{v+3}{v}\rfloor$ v-bit A/S	$d\log_2 d + \frac{3d}{2}$
	$(v+2+\frac{d+3v}{4}-1)$-bit SH & MR	$\lfloor\frac{v+2+\frac{d+3v}{4}-1}{v}\rfloor$ v-bit A/S	$\frac{d}{2}$
	$(v+2+\frac{d+v}{4}-1)$-bit SH & MR	$\lfloor\frac{v+2+\frac{d+v}{4}-1}{v}\rfloor$ v-bit A/S	$\frac{d}{2}$
	$(v+2+\frac{d}{2}-2)$-bit SH & MR	$\lfloor\frac{(v+2+(\frac{d}{2}-1)\log_2\omega)}{v}\rfloor$ v-bit A/S	$\frac{d}{2}(\log_2 d - 1)$
	$\lceil\frac{v+2}{\mu}\rceil$-input μ-bit A/S	$\lfloor\frac{v+2}{\mu}\rfloor$ μ-bit A/S	$\frac{d}{2}+1$

MUL, MR, SH, A/S represent multiplication, modular q reduction, shift, and 2-input addition or subtraction, respectively.

$|x| < 2^{2v}$, $x_0 - x_1$ is in range $[-2^v + 1, 2^v - 1]$. Thus, the residue only needs $v+1$ bits for storage including a sign bit. Since $x_0 - x_1$ can be performed by a subtractor in only one cycle, this method is applied to most of the modulo-q cases in the system. The overhead is that the following operators should support signed arithmetics, and therefore, 2's complement representation is preferred. Again, using the NLP form can save the correction step with negligible overhead.

Before performing division by r, the signed coefficients should be corrected to positive by performing Equation (7) again (cf. Section 5.4). Therefore, the data range in the entire FFTM^3 system is $[-2^v + 1, 2^v]$, and the data width is $v+2$ bits.

5.6 FFTM^3 Algorithm Modification

Using NLP form and carry-save accumulation will affect the parameter specification of FFTM^3. Specifically, employing NLP form will increase two more bits for each coefficient, and these $(\mu+2)$-bit coefficients will be taken into the following component-wise multiplication. Thus the 3rd parameter specification in Section 4 is revised to:

3) The greatest $b = 2^\mu$ such that $s \cdot 2^{2(\mu+2)+1} < q$.

Considering the inputs boundary, $g = xy < rn$ is required in MMM [3], [26]. Originally, $x, y < 2n$, but the NLP form introduces one more n range (cf. Section 5.3), so $x, y < 3n$ hence $t = xy < 9n^2 < rn$ should be satisfied. Consequently, $r > 9n$. To provide a conservative boundary, the 4th parameter specification is revised to:

4) $l = \mu s - 4$, where l is the maximum operand size of n.

When employing NLP form and carry-save accumulation, one could change the parameters in Table 3 according to the above specifications.

6 Efficient Parameter Selection Method for a Targeted Operand Size

After the introduction of Pseudo-Fermat number transform, the number of parameter set which can support the same operand size MMM will increase. Different parameter sets can be used to meet different design requirements. For example, if a compact design is required, the selection of small v would be a good choice. If a low area-latency product FFTM^3 is the target, then a reasonable selection of d and v is required. In order to achieve a better trade-off between area usage and latency, analyzing and comparing the complexity of FFTM^3 are preferred in the parameter set selection.

6.1 Area Complexity Evaluation of FFTM^3

The computation of FFTM^3 using Fermat and Pseudo-Fermat number transform includes component-wise multiplication, FFT and IFFT. Four basic operations, namely multiplication, addition/subtraction, shift, and modular reduction constitute these computations. The direct evaluation of area complexity of FFTM^3 would be the comparison of the number of these four operations.

The computational requirements of FFTM^3 by using Fermat and Pseudo-Fermat number transform are listed in Table 5. The area complexity of FFTM^3 is equal to the product of `equivalent operation complexity` and `number` `of equivalent operation`. The bit-level operation complexity for addition and subtraction is $O(l)$. In terms of multiplication, the complexity is determined according to the used multiplication algorithm as shown in Table 1.

In Table 5, when $\omega \neq \sqrt{2}$, according to Equation (5), the value of P_{jk} is in range $[0, \frac{d}{2}-1]$, thus the shift operation is range from $v+1$ to $v+1+(\frac{d}{2}-1)\log_2\omega$. Compared with FFT, IFFT has to deal with multiplication by d^{-1} and accumulation. For a fixed parameter set, the value of d^{-1} is also fixed. Thus the shift and modular reduction can be hardcoded as subtraction (modular reduction) on the designated bits.

When $\omega = \sqrt{2} = 2^{\frac{1}{2}}$, the multiplication by $\omega^{P_{jk}}$ can be decomposed into two conditions. If P_{jk} is even, multiply by $\omega^{P_{jk}}$ can be simply accomplished by shift operation. If P_{jk} is odd, as being pointed out by Nussbaumer [25] that $\sqrt{2} \equiv 2^{v/4}(2^{v/2}-1) \bmod q$, multiply by $\omega^{P_{jk}}$ will become two shifts and one subtraction as follows:

$$\begin{aligned} 2^{\frac{P_{jk}}{2}} &= 2^{\frac{P_{jk}-1}{2}+\frac{1}{2}} = 2^{\frac{P_{jk}-1}{2}} \cdot 2^{\frac{v}{4}}(2^{\frac{v}{2}}-1) \\ &= 2^{\frac{2(P_{jk}-1)+3v}{4}} - 2^{\frac{2(P_{jk}-1)+v}{4}} \quad (\bmod q). \end{aligned} \tag{12}$$

Note that modular reduction are required following the shift and subtraction, the multiplication by $\omega^{P_{jk}}$ would become two shifts, one subtraction, and three modular reductions. Because the maximum value of P_{jk} is $\frac{d}{2}-1$, the maximum value of $\omega^{P_{jk}}$ would become

$$2^{\frac{2(\frac{d}{2}-1-1)+3v}{4}} - 2^{\frac{2(\frac{d}{2}-1-1)+v}{4}} = 2^{\frac{d+3v}{4}-1} - 2^{\frac{d+v}{4}-1}.$$

Carefully examining Equation (5) one can observe that odd P_{jk} only appear when $j=0$ (stage 0 of FFT/IFFT). Thus the multiplication by odd P_{jk} only appears $\frac{d}{4}$ times in each FFT/IFFT. This indicates that using $\omega = \sqrt{2}$ would not increase too much workload for the computation, but are able to double the maximum operand size of the parameter set within a fixed q.

6.2 Efficient Parameter Set Selection method for FFTM^3

Given a target operand size l, the word size μ can be found after the confirmation of d. With these informations, the smallest q can be calculated. The parameter selection method for a target operand size l is shown below.

1) Select power of 2 number d from 2 to $l+4$;
2) Let $s = d/2$, calculate $\mu = \lceil\frac{l+4}{s}\rceil + 2$;
3) Determine τ_1, τ_2 by $\tau_1 = \log_2 d - 1$ and $\tau_2 = \log_2 d - 2$, respectively;
4) Calculate $v_{min} = 2\mu + \log_2 s + 1$ due to $q = 2^v + 1 > s2^{2\mu+1} = 2^{2\mu+\log_2 s+1}$;
5) For the positive τ_i $(i = 1, 2)$, compute $\delta_i = \lceil\frac{v_{min}}{2^{\tau_i}}\rceil$. Discard the δ_2 parameter set if $\delta_2 \neq 1$;
6) For the existent δ_i $(i = 1, 2)$, resize $v_i = \delta_i 2^{\tau_i}$, $q_i = 2^{v_i} + 1$;
7) For the existent δ_i $(i = 1, 2)$, determine $w_1 = 2^{\delta_1}$ and $w_2 = 2^{2^{\tau_2-2}}(2^{2^{\tau_2-1}}-1)$.

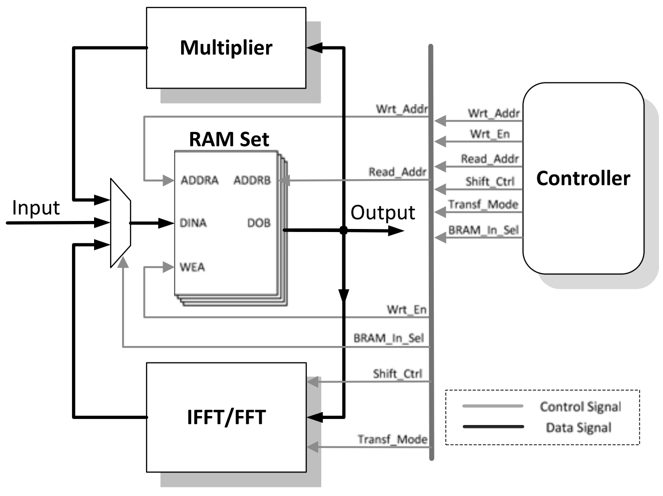


Fig. 4. The top level architecture of FFTM^3. *shift_ctrl* signals control the bit shift operation of FFT/IFFT. *Transf_Mode* select the transformation mode between FFT and IFFT.

8) Store the existent parameter sets $(d, s, \mu = \mu - 2, q_i = 2^{v_i}+1, \omega_i, l = \mu s - 4)$ for $i = 1, 2$. If $\delta_2 = 1$ stop the selection process, else go to Step 1.

The selection process is terminated when $\delta_2 = 1$. This is because after δ_2 is fixed equal to 1, with the increasing of d, v will be larger than the value in the previous parameter set. Therefore, the complexity for these new parameter sets will not be lower than the previous set. Hence, they can be eliminated from the comparison.

By using the proposed method, one could select the parameter set by comparing the area complexity. When the smallest area-latency product is the target, the cycle number and the clock period are also needed to be concerned. In this case, the comparison would be carried out on the product of `area complexity`, `cycle number`, and `clock period`.

7 Pipelined Architecture for FFTM^3

In FFTM^3 algorithm, the same computation pattern (component-wise multiplication, IFFT, mod/div, and FFT) repeats for three times. Thus, a cost-effective architecture can be designed to perform this patten by reusing this architecture for three times. Furthermore, since the dataflows of FFT and IFFT are similar, they can share the same computation resources. The top level architecture of FFTM^3 is shown in Fig. 4. The multiplier and FFT/IFFT modules are working in pipelined. Different operand sizes of FFTM^3 range from 1,024-bit to 15,484-bit are designed.

Considering the component-wise multiplication in FFTM^3, SSA method can be applied recursively when the multiplier's operand size is large enough. However, when the operand size is small, the schoolbook method or Karatsuba algorithm is simpler and more efficient. This idea was also noted by [29] that the optimal multiplication algorithm might use different method in different level of computation.

In this work, Karatsuba's method [12], [18] is used to build the $(v+2)$-bit multiplier. The architecture of the Karatsuba multiplier is shown in Fig. 5. In order to compute a 2γ-bit multiplication, one only needs two γ-bit and one $(\gamma+1)$-bit multiplications instead of four γ-bit

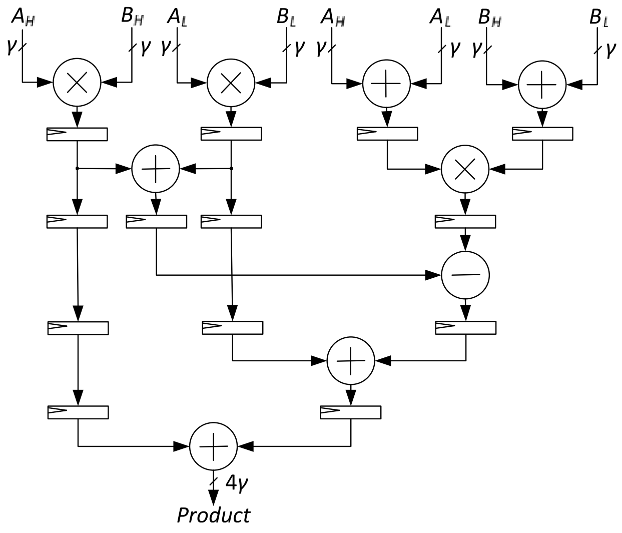


Fig. 5. Proposed architecture of the 2γ-bit Karatsuba multiplier.

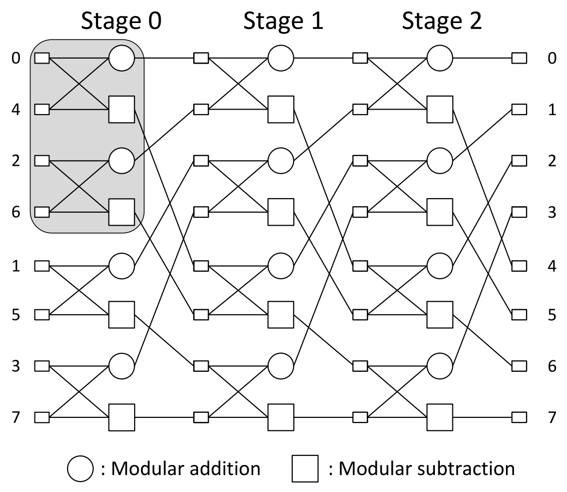


Fig. 6. Example architecture of a length-8 constant geometry FFT. The shadow area is used as the primitive for the sub-stage FFT/IFFT architecture design.

multiplications. One can also apply Karatsuba multiplication recursively using divide-and-conquer approach.

7.1 Architecture for FFT/IFFT

FFT/IFFT is quite mature in signal processing area [30]. However, to our knowledge, there are only a few research works on the architectures for NTT [31], [32], [33], and they are either area-consuming or slow. Thus the design of number theoretic FFT/IFFT architecture which fits for FFTM^3 is required.

In practice, area and speed trade-off are needed to be concerned in the design of the FFT/IFFT module. There are two extreme cases which are either a single butterfly structure, or unlooping all the stages to build the whole butterfly network. However, the former one is too slow, while the later one consumes too much hardware resources, and the long wire routing of such a large design will drag down the operating frequency.

In order to balance the scenarios mentioned above, a length-4 sub-stage FFT/IFFT architecture is designed as shown in the shadowed area in Fig. 6. This sub-stage module can pipeline the intra-stage operations, and be reused for the whole FFT/IFFT computation. The length-4 sub-stage FFT/IFFT architecture is shown in Fig. 7.

As can be seen in Fig. 7, the length-4 sub-stage FFT/IFFT module has two butterfly structures which can handle four coefficients at each cycle. The shift operators are responsible for multiplying $\omega^{P_{jk}}$ to x_{2k+1} in Equation (5). According to the stage number of FFT/IFFT and the coefficient order, shift operators can perform the corresponding multiplication operation.

In terms of IFFT, the coefficients should be multiplied by d^{-1} and added up after the last stage butterfly computation. The multiplication by d^{-1} is performed by shift operation because d^{-1} is a power of 2 (cf. Section 2.2). In our design, parallel adders are used for a fast carry-save accumulation as shown in Fig. 3. Since there are two coefficients output from IFFT in each cycle, two B-input μ-bit adders ($B = \lceil \frac{v+2}{\mu} \rceil$) are designed to work in parallel.

The ϵ is needed to correct the division-by-r result. Still taking $B = 3$ as an example, in Equation (11), the carry-in for the two highest bits could only be 0, 1, or 2, and the sum with carry-in should be a multiple of 4 to satisfy Fact 1. Therefore, $\lfloor 4z_{(s-1)0}/b \rfloor + \lfloor 4z_{(s-2)1}/b \rfloor$ cannot be 1 or 5. Let a_1,

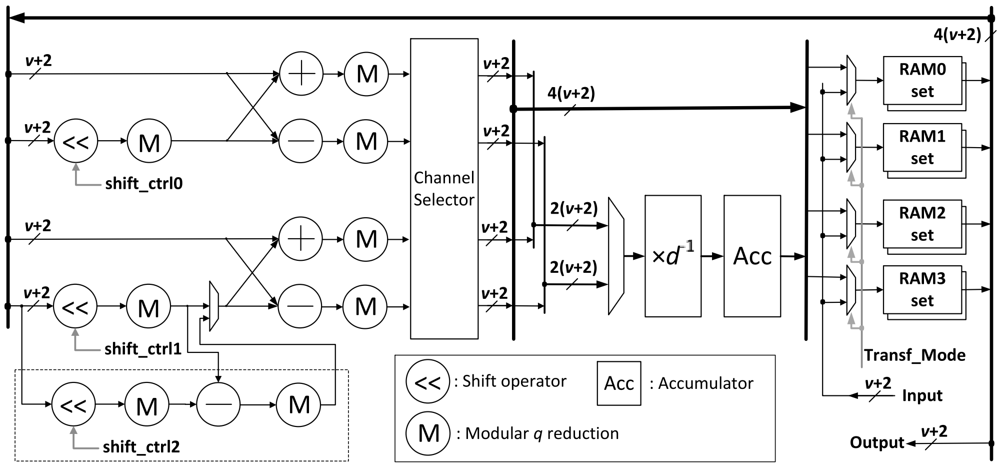


Fig. 7. Detailed architecture of our FFT/IFFT module. Registers are existed between each two operators, we omitted them in this figure for simplicity. *shift_ctrlX* signals control the bit shift operation of the shift operators. *Transf_Mode* select different data path to finish FFT and IFFT computation. The operators in dash line bounding box are for the FFTM^3 which $\omega = 2^{2^{\tau-2}}(2^{2^{\tau-1}} - 1)$.

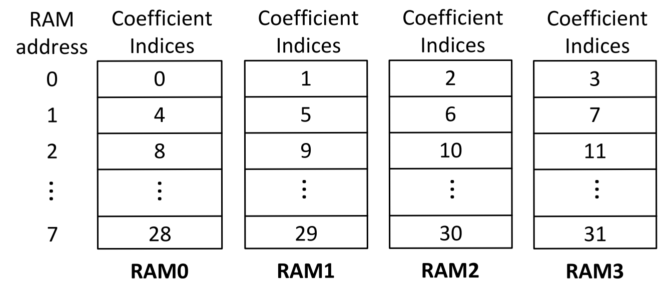


Fig. 8. An example of a length-32 FFT/IFFT coefficients storage situation in BRAMs.

a_0, b_1, b_0 be the two most significant bits of $z_{(s-1)0}$ and $z_{(s-2)1}$, respectively, then the logic for ϵ can be simplified as:

$$\begin{aligned}\epsilon[0] &= (a_1 \oplus a_0) \vee (b_1 \oplus b_0) \vee (a_1 \oplus b_1),\\ \epsilon[1] &= a_1 \wedge a_0 \wedge b_1 \wedge b_0,\end{aligned}$$

where $\wedge, \vee, \oplus$ are *and*, *or*, *xor* operators, respectively.

7.2 Memory Management

The data-width in FFTM^3 algorithm is $(v+2)$-bit, and the total size of the coefficients is $(v+2)\cdot d$ bits, which is quite a big size when v and d are large. Therefore, these coefficients are stored in memory blocks rather than registered on the fly.

Simple dual-port RAMs (RAM), which have one dedicated read port and one dedicated write port, are used for coefficients storage. As the FFT/IFFT module has four inputs, we employ four RAMs so that all the four inputs for FFT/IFFT can be provided in one cycle. An example of the coefficients storage situation for length-32 FFT/IFFT case is depicted in Fig. 8. Every four successive coefficients are distributed into the four RAMs in the same address. Therefore, by using one address signal we can read out all the needed coefficients for the parallel substage FFT/IFFT computation.

The write control is more complex because after each stage of FFT/IFFT, the order of the coefficients is shuffled. We call `RAM0` and `RAM1` the *higher half* RAMs, `RAM2` and `RAM3` the *lower half* ones. The input/output sequence for a length-32 FFT/IFFT is shown in Fig. 9. For instance, in Time 0, Coefficients 0-3 are fetched from RAMs 0–3. After i cycles, the FFT/IFFT module finishes the processing. However, the outputs are Coefficients x_0, x_1, x_{16} and x_{17}. Note that Coefficients x_0 and x_{16} belong to RAM 0 and Coefficients x_1 and x_{17} belong to RAM 1, a direct write back will produce write collision. The similar situation also happens for the outputs of other time slots.

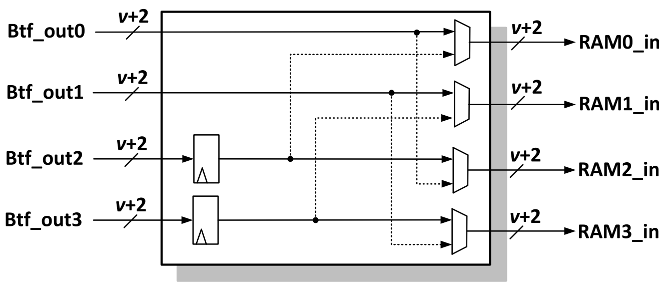


Fig. 10. Architecture of Channel Selector. At each cycle, only solid line or dash line channels will output data. MUXs are used to select the channels of solid lines and dash lines. Control signals are omitted for simplicity.

To wire the coefficients back to the appropriate RAMs without collision, the channel selector is designed as shown in Fig. 10. In Time $i+0$, Coefficients x_0-x_1 pass through while Coefficients x_{16}-x_{17} are stored in the register. In Time $i+1$, Coefficients x_2-x_3 are switched to the lower half outputs, and the Coefficients x_{16}–x_{17} in the registers are switched back to the higher half which connected to RAMs 0-1. At the same time, Coefficients x_{18}-x_{19} arrive at the registers. One cycle later, they pass to the lower half outputs along with Coefficients x_4-x_5 passing to the higher half. The channel selector works so on, with one cycle overhead, four coefficients which are shown as the bounding box in Fig. 9, can be written to their corresponding RAMs.

When the pipeline depth of FFT/IFFT structure i is smaller than $d/8$, a racing problem will occur. Take Fig. 9 as an example. Coefficients x_{16}-x_{17} will not read out of RAMs until Time 4. However, the new Coefficients x_{16}–x_{17} for next stage will arrive at Time $i+1$, which is on or before Time 4 if $i < \frac{32}{8} = 4$. Since the storage location for these coefficients is overlapped, the storage of the new coefficients will overwrite the coefficients of current stage. In order to tackle this problem without adding pipelined bubbles, one more set of RAMs is appended and the ping-pong alternative storage mechanism is employed.

One may argue that the in-place FFT algorithm can handle the write collision. However, the in-place FFT algorithm is not suitable to design a sub-stage architecture; each substage of in-place FFT has different butterfly network, the circuit for the butterfly network switch is more complex than the channel selector.

7.3 Cycle Requirement Analysis

By using the proposed architecture, the pipelined delay i of FFT/IFFT module is 10 cycles. For each stage except the last stage of FFT/IFFT, the cycle requirement from the input to the output is $10 + \frac{d}{4} - \frac{d}{8} - 1$ while outputting d coefficients

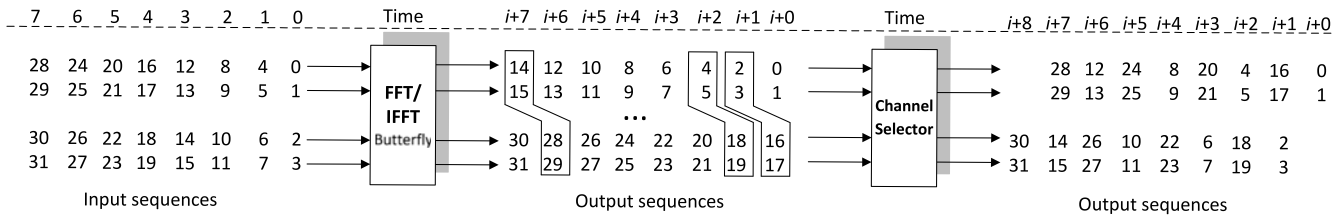


Fig. 9. Input and output orders of a length-32 FFT/IFFT sub-stage architecture. Four successive coefficients are inputed at each clock cycle. After i clock cycles the coefficients will be outputted. Here i is also known as the pipeline depth for the FFT/IFFT architecture.

TABLE 6
Cycle Requirement of the Proposed FFTM^3 Architecture

		$d < 128$ and $\omega \neq \sqrt{2}$	$d < 128$ and $\omega = \sqrt{2}$	$d \geq 128$ and $\omega \neq \sqrt{2}$	$d \geq 128$ and $\omega = \sqrt{2}$
Component-wise multiplication		$Base\_mult\_delay + d - \frac{d}{4} + 1$			
FFT	($\log_2 d$ - 1)-th stage	$10 + \frac{d}{4} - \frac{d}{16} - 1$	$13 + \frac{d}{4} - \frac{d}{16} - 1$	$10 + \frac{d}{4} - \frac{d}{16} - 1$	$13 + \frac{d}{4} - \frac{d}{16} - 1$
	Other stages	$10 + \frac{d}{4} - \frac{d}{8} - 1$	$10 + \frac{d}{4} - \frac{d}{8} - 1$	$\frac{d}{4}$	$\frac{d}{4}$
IFFT	($\log_2 d$ - 1)-th stage	$14 + \frac{d}{4} - \frac{d}{16} - 1$	$17 + \frac{d}{4} - \frac{d}{16} - 1$	$14 + \frac{d}{4} - \frac{d}{16} - 1$	$17 + \frac{d}{4} - \frac{d}{16} - 1$
	Other stages	$10 + \frac{d}{4} - \frac{d}{8} - 1$	$10 + \frac{d}{4} - \frac{d}{8} - 1$	$\frac{d}{4}$	$\frac{d}{4}$

Base_mult_delay means the cycle delay of the base multiplier.

TABLE 7
Virtex-6 Implementation Results and Comparisons of the FFTM^3 Architecture

Maximun operand size (bit)	v	d	μ	LUTs	Slice	DSP48	RAMB36 / RAM18	Cycles	Period (ns)	Latency (μs)	Area-latency product Implemented : Theoretic (LUTs $\times$ μs) : ($\times 10^7$)
1,028	272	16	129	25,309	8,702	135	44/0	387	9.37	3.80	96,174 : 3.96
1,036	144	32	65	13,573	4,786	54	22/11	553	6.59	3.87	52,527 : 2.11
1,052	96	64	33	7,534	2,519	36	11/11	840	5.87	4.96	37,368 : 1.56
1,084	64	128	17	4,818	1,483	9	11/0	1,440	5.29	7.57	36,472 : 1.86
2,060	272	32	129	25,837	8,366	135	44/0	546	8.8	4.80	124,017 : 5.68
2,076	160	64	65	14,895	5,534	54	22/11	837	6.6	5.52	82,220 : 3.64
2,108	128	128	33	11,824	4,071	27	22/0	1,701	6.0	10.21	120,723 : 6.08
2,172	64	256	17	5,737	2,017	9	11/0	3,633	5.09	18.49	106,077 : 4.36
3,100	224	64	97	21,672	7,147	108	33/11	843	8.0	6.74	146,069 : 7.20
3,132	128	128	49	12,147	4,062	27	22/0	1,701	6.09	10.36	125,842 : 5.92
3,196	128	256	25	11,728	3,562	27	22/0	3,693	6.19	22.86	268,102 : 13.7
3,196	64	256	25	5,835	1,977	9	11/0	3,633	5.09	18.49	107,889 : 4.21
4,124	288	64	129	27,839	9,530	135	44/11	846	9.2	7.78	216,587 : 9.29
4,156	192	128	65	18,449	6,351	108	33/0	1,710	8.0	13.68	252,382 : 12.41
4,220	128	256	33	11,919	3,892	27	22/0	3,693	6.3	23.27	277,355: 13.21
7,740	256	128	121	30,230	9,281	81	44/0	1,713	8.60	14.73	445,287 : 16.75
15,484	256	256	121	30,405	9,425	81	44/0	3,654	8.77	32.05	974,480 : 36.27

TABLE 8
Virtex-II Implementation Results and Comparisons of the Modular Multipliers

Max. operand size (bit)	Design	LUT	Slice	Latency (μs)	Area-latency product		Area-latency product improvement (%)
					(LUT $\times\mu s$)	(Slice $\times\mu s$)	
1,024	[5] (5 to 2)	-	10,332	10.08	-	104,146	26.2
1,024	[10] (radix-2, $\omega = 32$)	5,310	-	10.09	53,577	-	−36.7
1,084	Our design	5,843	5,310	14.48	84,606	76,888	-
2,048	[5] (5 to 2)	-	20,986	22.76	-	477,641	48.5
2,048	[10] (radix-2, $\omega = 32$)	10,587	-	20.68	218,939	-	−15.5
2,076	Our design	18,200	17,297	14.23	258,986	246,136	-
3,072	[10] (radix-2, $\omega = 32$)	15,197	-	30.94	470,195	-	34.2
3,196	Our design	7,210	5,820	42.91	309,381	249,736	-
4,096	[10] (radix-2, $\omega = 32$)	19,621	-	41.85	821,138	-	18.08
4,124	Our design	38,972	37,086	17.26	672,656	640,104	-

requires $d/4$ cycles. The cycle requirement for each stage of FFT/IFFT (except the last stage) is the larger number between these two. Therefore, when $d < 72$ the cycle requirement equals to $10 + \frac{d}{4} - \frac{d}{8} - 1$ while it is $d/4$ when $d \geq 72$. Due to the fact that d is power of 2 number, the condition will switch to $d < 128$ and $d \geq 128$.

In terms of the cycle requirement of the last stage (($\log_2 d$ - 1)th stage) of IFFT, the multiplication by d^{-1} and the accumulation consume four more cycles. Additionally, when $\omega = 2^{2^{\tau-2}}(2^{2^{\tau-1}} - 1) = \sqrt{2}$, three more cycles are required in the last stage of FFT/IFFT. The cycle requirement of the proposed FFTM^3 architecture is summarized in Table 6.

8 Implementations and Comparisons

The FFTM^3 architectures are described in Verilog-HDL with synthesis and place & route by Xilinx ISE 13.3. The proposed architectures are implemented on a Xilinx Virtex-6 (xc6vlx130t-1) FPGA. In the Virtex-6 device, each DSP48E1 has a 18-bit signed multiplier, and we use it as the base multiplier to build the $(v+2)$-bit Karatsuba pipelined multiplier. The base multipliers in DSP48E1s are pipelined with the optimal stages in order to increase the frequency.

The post-place-and-route results and comparisons are listed in Table 7. In this paper, we targeted to find out the parameter set which has the lowest area-latency product for each targeted operand size FFTM^3. The number of both area-latency product complexity analysis and the real implementation results are shown in Table 7. In terms of the area-latency product complexity analysis, the number is calculated by the product of area complexity (cf. Section 6.1), cycle number (cf. Section 7.3), and clock period (cf. Section 7.3). It can be found that complexity analysis matches with the real implementation results.

In order to have a fair comparison with previous works, we also implemented FFTM^3 on a Xilinx Virtex-II-4 FPGA using Xilinx ISE 10.1. We compare our work with the Montgomery modular multiplication architectures from [5], [10] in Table 8. The improvement on area-latency product is also shown in Table 8.

In order to show the growth trend of the area-latency product of MMM in [10] and FFTM^3 intuitively, we depicted the data in Fig. 11. As shown in Fig. 11, the growth rate of FFTM^3 is slower than that of the MMM in [10]. The FFTM^3 outperforms the MMM in [10] when the operand size is large enough. Specifically, at 1,024-bit level, our design has no advantage compared with the results in [10]. But when the operand size reach 3,196-bit and 4,124-bit, the FFTMMMs have 34.2 and 18.1 percent improvement, respectively, when compared with the 3,072-bit and 4,096-bit MMM in [10];

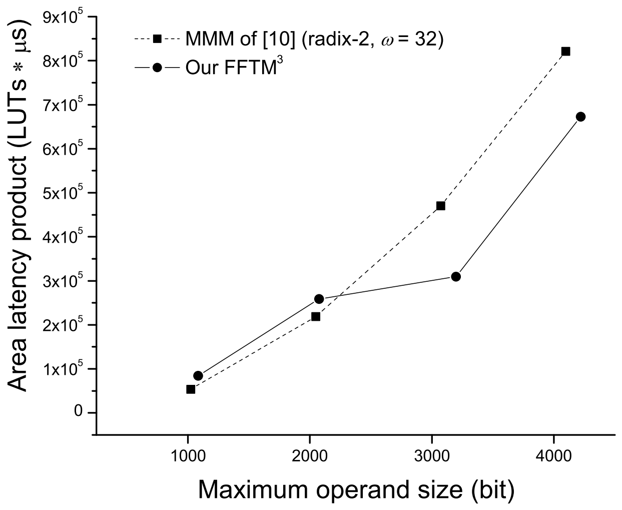


Fig. 11. Area-latency product comparison with [10].

The performance comparisons in terms of latency are shown in Table 9. Compared with [10] and [11], our 4,156-bit FFTM^3 outperforms their 4,096-bit design by approximately 143 and 326 percent, respectively. The speedup

TABLE 9
Latency Comparison of Hardware and Software Modular Multipliers

Max. oper. size (bit)	Design	Device	LUT/Slice	Cycle red. (%)	Cycle	Period (ns)	Latency (μs)	Speedup (%)
1,024	[5] (5 to 2)	Virtex-II-4	-/10,332	1,025	18.0	9.83	10.08	−9.0
1,024	[10] (radix-2, $\omega = 32$)	Virtex-II-4	5,310/-	1,056	20.5	9.55	10.09	−8.9
1,052	Our design	Virtex-II-4	9,880/8,489	840	-	13.19	11.08	-
2,048	[5] (5 to 2)	Virtex-II-4	-/20,986	2,050	59.2	11.10	22.76	59.9
2,048	[10] (radix-2, $\omega = 32$)	Virtex-II-4	10,587/-	2,112	60.4	9.79	20.68	45.3
2,076	Our design	Virtex-II-4	18,200/17,297	837	-	17.00	14.23	-
3,072	[10] (radix-2, $\omega = 16$)	Virtex-II-4	16,331/-	3,264	74.2	9.55	31.17	85.2
3,072	[10] (radix-2, $\omega = 32$)	Virtex-II-4	15,197/-	3,168	73.4	9.76	30.94	83.8
3,100	Our design	Virtex-II-4	26,746/25,578	843	-	19.96	16.83	-
4,096	[10] (radix-2, $\omega = 16$)	Virtex-II-4	21,139/-	4,352	80.6	9.69	42.20	144.5
4,096	[10] (radix-2, $\omega = 32$)	Virtex-II-4	19,621/-	4,224	80.0	9.91	41.85	142.5
4,124	Our design	Virtex-II-4	38,972/37,086	846	-	20.40	17.26	-
1,024	[8] (3 Threads MMM)	Intel Xeon X5650	-/-	9,654*	95.6	0.376	3.63	40.2
1,024	[11]	90nm CMOS	-/-	1,036*	62.6	1.68	1.74	−32.8
1,028	Our design	Virtex-6-3	25,143/8,706	387	-	6.69	2.59	-
2,048	[8] (4 Threads MMM)	Intel Xeon X5650	-/-	13,138*	95.8	0.376	4.94	35.3
2,048	[11]	90nm CMOS	-/-	3,881*	85.9	1.68	6.52	78.6
2,060	Our design	Virtex-6-3	25,587/8,860	546	-	6.69	3.65	-
3,072	[8] (4 Threads MMM)	Intel Xeon X5650	-/-	17,393*	95.2	0.376	6.54	31.6
3,072	[11]	90nm CMOS	-/-	8,708*	90.3	1.68	14.63	94.4
3,100	Our design	Virtex-6-3	21,349/6,788	843	-	5.89	4.97	-
4,096	[8] (6 Threads MMM)	Intel Xeon X5650	-/-	23,191*	96.4	0.376	8.72	43.4
4,096	[11]	90nm CMOS	-/-	15,429*	94.5	1.68	25.92	326.3
4,124	Our design	Virtex-6-3	29,345/10,200	846	-	7.19	6.08	-

* *Estimated number calculated by the quotient of latency and period.*

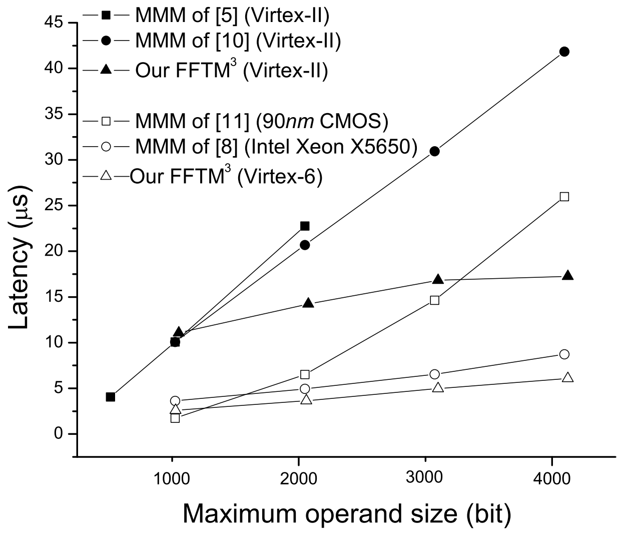


Fig. 12. Latency comparison with hardware and software implementations from [5], [8], [10], [11].

mainly comes from the low complexity of the FFTM^3 algorithm and the pipelined design, which reduce the computation clock cycles significantly.

We also compare our work with the software design in Giorgi et al.'s work [8]. Note that Giorgi's work aims at low latency and the implementations are carried out on two powerful Intel Xeon X5650 Westmere processors which the operating frequency reaches 2.66 GHz. Since our hardware design is targeted for small area-latency product, parallelism is not fully exploited. However, our design can still have a latency reduction of approximately 38 percent on average when compared with the fastest MMM results in [8].

The latency growth rate of MMM [5], [8], [10], [11], and FFTM^3 are described in Fig. 12. As shown in Fig. 12, the latency of these MMMs increase linearly. However, the slope of our FFTM^3 is smaller than the others, which implies that the performance of FFTM^3 will be more obvious for the larger operand sizes.

9 Conclusions and Future Works

In this paper, the FFT-based Montgomery modular multiplication algorithm is improved by using carry-save arithmetic and pre-computation techniques. The pseudo-Fermat number transform is introduced to further enrich the supported operand size of FFTM^3. The parameter requirements of the improved FFTM^3 are analyzed, and a fast method for the selection of targeted parameter sets is proposed. Furthermore, we have designed a hardware pipelined architecture of the FFTM^3 targeting for low area-latency product. The results show that the proposed FFTM^3 architecture outperforms the state of the arts for large operand sizes design.

Notice that McLaughlin's algorithm [21] has a lower theoretic computation complexity than the FFTM^3, which might be a good candidate to speedup the large size modular multiplication. As the hardware performance of the McLaughlin's algorithm is still unknown, we conduct a preliminary study on McLaughlin's algorithm from hardware design aspects as shown in the Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2015.2417553. We also compare both the cycle requirement and the architecture design between McLaughlin's algorithm and FFTM^3.

The comparisons reveal that when the design following similar architectures, McLaughlin's algorithm has advantage over FFTM^3 in terms of cycle number when the NTT length is large enough. From the hardware design point of view, McLaughlin's algorithm has the serial accumulation and a more complex dataflow, which could be the bottleneck for a parallel or compact architecture design. Therefore, it is worthwhile to implement McLaughlin's algorithm and compare the results with FFTM^3 to find out the performance crosspoint for a more cost effective modular multiplier design.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive suggestions and comments on our paper. Especially for the reviewer to point out the future research direction on the hardware design for the McLaughlin's algorithm. This work was partly supported by the Research Grant Council of the Hong Kong Special Administrative Region, China (Projects No. CityU 123612, CityU 111913), and Croucher Startup Allowance (Project No. 9500015). Donald Donglong Chen is the corresponding author.

References

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.
[2] "Recommendation for key management," NIST, Tech. Rep. Special Publication 800-57, Part 1, Rev. 3, 2012.
[3] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, 1985.
[4] A. Tenca and Ç. K. Koç, "A scalable architecture for modular multiplication based on Montgomery's algorithm," *IEEE Trans. Comput.*, vol. 52, no. 9, pp. 1215–1221, Sep. 2003.
[5] C. McIvor, M. McLoone, and J. McCanny, "Modified Montgomery modular multiplication and RSA exponentiation techniques," in *Proc. IEE Proc.—Comput. Digital Techn.*, 2004, vol. 151, no. 6, pp. 402–408.
[6] D. S. Phatak and T. Goff, "Fast modular reduction for large wordlengths via one linear and one cyclic convolution," in *Proc. Comput. Arithmetic*, 2005, pp. 179–186.
[7] J. David, K. Kalach, and N. Tittley, "Hardware complexity of modular multiplication and exponentiation," *IEEE Trans. Comput.*, vol. 56, no. 10, pp. 1308–1319, Oct. 2007.
[8] P. Giorgi, L. Imbert, and T. Izard, "Parallel modular multiplication on multi-core processors," in *Proc. 21st IEEE Symp. Comput. Arithmetic*, Apr. 2013, pp. 135–142.
[9] M. Shieh and W. Lin, "Word-based Montgomery modular multiplication algorithm for low-latency scalable architectures," *IEEE Trans. Comput.*, vol. 59, no. 8, pp. 1145–1151, Aug. 2010.
[10] M. Huang, K. Gaj, and T. El-Ghazawi, "New hardware architectures for Montgomery modular multiplication algorithm," *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 923–936, Jul. 2011.
[11] W.-C. Lin, J.-H. Ye, and M.-D. Shieh, "Scalable Montgomery modular multiplication architecture with low-latency and low-memory bandwidth requirement," *IEEE Trans. Comput.*, vol. 63, no. 2, pp. 475–483, Feb. 2014.
[12] A. Karatsuba and Y. Ofman,"Multiplication of multidigit numbers on automata," *Soviet Phys. Doklady*, vol. 7, no. 7, pp. 595–596, 1963.
[13] S. A. Cook, "On the minimum computation time of functions," Ph.D. dissertation, Dept. Math., Harvard Univ., Cambridge, MA, USA, 1966.
[14] A. Schönhage and V. Strassen, "Schnelle multiplikation großer zahlen," *Computing*, vol. 7, pp. 281–292, 1971.

[15] M. Fürer, "Faster integer multiplication," in *Proc. 39th Annu. ACM Symp. Theory Comput.*, 2007, pp. 57–66.

[16] T. Granlund and the GMP development team, "The GNU multiple precision arithmetic library (GMP)," [Online]. Available: www.gmplib.org, 2014.

[17] G. Chow, K. Eguro, W. Luk, and P. Leong, "A Karatsuba-based Montgomery multiplier," in *Proc. Field Programmable Logic Appl.*, 2010, pp. 434–437.

[18] M. K. Jaiswal and R. C. C. Cheung, "Area-efficient architectures for large integer and quadruple precision floating point multipliers," in *Proc. Field-Programmable Custom Comput. Mach.*, 2012, pp. 25–28.

[19] G. Saldamlı and Ç. K. Koç, "Spectral modular exponentiation," in *Proc. Comput. Arithmetic*, 2007, pp. 123–132.

[20] Ç. K. Koç, T. Acar, and S. B. Kaliski, "Analyzing and comparing Montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, Jun. 1996.

[21] P. B. Mclaughlin,"New frameworks for Montgomery's modular multiplication method," *Math. Comput.*, vol. 73, no. 246, pp. 899–906, 2003.

[22] R. Creutzburg and M. Tasche, "Number-theoretic transforms of prescribed length," *Math. Comput.*, vol. 47, pp. 693–701, 1986.

[23] J. M. Pollard, "The fast Fourier transform in a finite field," *Math. Comput.*, vol. 25, pp. 365–374, 1971.

[24] J. Cooley and J. Turkey,"An algorithm for the machine computation of complex Fourier series," *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.

[25] H. J. Nussbaumer, *Fast Fourier Transform and Convolution Algorithms*. New York, NY, USA: Springer, 1982.

[26] C. D. Walter,"Montgomery exponentiation needs no final subtractions," *Electron. Lett.*, vol. 35, no. 21, pp. 1831–1832, 1999.

[27] N. Smart, "ECRYPT II yearly report on algorithms and keysizes (2011–2012)," ECRYPT II, Tech. Rep. ICT-2007-216676, 2012.

[28] F. de Dinechin, H. D. Nguyen, and B. Pasca, "Pipelined FPGA adders," in *Proc. Field Programmable Logic Appl.*, 2010, pp. 422–427.

[29] D. J. Bernstein. (2001). Multidigit multiplication for mathematicians [Online]. Available: http://cr.yp.to/papers.html#m3

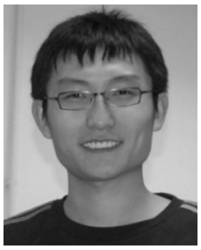
[30] Y.-N. Chang and K. Parhi, "An efficient pipelined FFT architecture," *IEEE Trans. Circuits Syst. II: Analog Digital Signal Process.*, vol. 50, no. 6, pp. 322–325, Jun. 2003.

[31] J. McClellan, "Hardware realization of a Fermat number transform," *IEEE Trans. Acoust., Speech Signal Process.*, vol. 24, no. 3, pp. 216–225, Jun. 1976.

[32] G. X. Yao, R. C. C. Cheung, Ç. K. Koç, and K. F. Man, "Reconfigurable number theoretic transform architectures for cryptographic applications," in *Proc. Field-Programmable Technol.*, 2010, pp. 308–311.

[33] D. Chen, N. Mentens, F. Vercauteren, S. Roy, R. Cheung, D. Pao, and I. Verbauwhede, "High-speed polynomial multiplication architecture for ring-LWE and SHE cryptosystems," *IEEE Trans. Circuits Syst. I: Regular Papers*, vol. 62, no. 1, pp. 157–166, Jan. 2015.

Donald Donglong Chen received the BEng degree in electronic and information engineering from Wuhan University of Technology in 2011. He is currently working toward the PhD degree at the Department of Electronic Engineering, City University of Hong Kong. His research interests include cryptographic algorithm optimization in hardware and high-speed digital system design, especially in field programmable gate array (FPGA). He is a student member of the IEEE.


Gavin Xiaoxu Yao received the bachelor's degree in control science and engineering from the Huazhong University of Science and Technology, Wuhan, China, in 2009, and the PhD degree in electronic engineering from the City University of Hong Kong in 2014. His research interests include hardware/algorithm codesign for public-key cryptography, fast arithmetic using residue number system, and field programmable gate array (FPGA).

Ray C.C. Cheung received the BEng and MPhil degrees in computer engineering and computer science and engineering at the Chinese University of Hong Kong (CUHK), Hong Kong, in 1999 and 2001, respectively, and the PhD degree and DIC in computing at Imperial College London, London, United Kingdom, in 2007. After completing the PhD work, he received the Hong Kong Croucher Foundation Fellowship for his postdoctoral study in the Electrical Engineering Department, University of California, Los Angeles (UCLA). In 2009, he worked as a visiting research fellow in the Department of Electrical Engineering, Princeton University, Princeton, NJ. Currently, he is an assistant professor in the Department of Electronic Engineering, City University of Hong Kong (CityU). He is the author of more than 40 journal papers and more than 50 conference papers. His research team, CityU Architecture Lab for Arithmetic and Security (CALAS), focuses on the following research topics: reconfigurable trusted computing, applied cryptography, and high-performance biomedical VLSI designs. He is a member of the IEEE.

Derek Pao received the BSc(Eng) degree in electrical engineering from the University of Hong Kong, and the master's and PhD degrees in computer science from Concordia University, Canada. He is an associate professor with the Electronic Engineering Department, City University of Hong Kong. His research interests include hardware architectures for network processing, and high speed pattern matching for network and system security. He is a member of the IEEE.

Çetin Kaya Koç received the PhD degree in electrical & computer engineering from the University of California Santa Barbara in 1988. His research interests are in cryptographic hardware and embedded systems, secure hardware design, side-channel attacks and countermeasures, algorithms and architectures for computer arithmetic and finite fields. He is the cofounder of the Workshop on Cryptographic Hardware and Embedded Systems, and the founding editor-in-chief of the *Journal of Cryptographic Engineering*. He has also been in the editorial boards of *IEEE Transactions on Computers* (2003-2008) and *IEEE Transactions on Mobile Computing* (2003-2007). Furthermore, he was a guest coeditor of April 2003 & November 2008 issues of the *IEEE Transactions on Computers*. He is the coauthor of the three books *Cryptographic Algorithms on Reconfigurable Hardware*, *Cryptographic Engineering*, and *Open Problems in Mathematics and Computational Science*, published by Springer. In 2007, he was elected as an IEEE fellow for his contributions to cryptographic engineering.