This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2601334, IEEE Transactions on Computers

Area-time Efficient Architecture of FFT-based Montgomery Multiplication

Wangchen Dai, Donald Donglong Chen, *Student Member, IEEE,*
Ray C. C. Cheung, *Member, IEEE,* and Çetin Kaya Koç, *Fellow, IEEE*

Abstract—The modular multiplication operation is the most time-consuming operation for number-theoretic cryptographic algorithms involving large integers, such as RSA and Diffie-Hellman. Implementations reveal that more than 75% of the time is spent in the modular multiplication function within the RSA for more than 1024-bit moduli. There are fast multiplier architectures to minimize the delay and increase the throughput using parallelism and pipelining. However such designs are large in terms of area and low in efficiency. In this paper, we integrate the fast Fourier transform (FFT) method into the McLaughlin's framework, and present an improved FFT-based Montgomery modular multiplication (MMM) algorithm achieving high area-time efficiency. Compared to the previous FFT-based designs, we inhibit the zero-padding operation by computing the modular multiplication steps directly using cyclic and nega-cyclic convolutions. Thus, we reduce the convolution length by half. Furthermore, supported by the number-theoretic weighted transform, the FFT algorithm is used to provide fast convolution computation. We also introduce a general method for efficient parameter selection for the proposed algorithm. Architectures with single and double butterfly structures are designed obtaining low area-latency solutions, which we implemented on Xilinx Virtex-6 FPGAs. The results show that our work offers a better area-latency efficiency compared to the state-of-the-art FFT-based MMM architectures from and above 1024-bit operand sizes. We have obtained area-latency efficiency improvements up to 50.9% for 1024-bit, 41.9% for 2048-bit, 37.8% for 4096-bit and 103.2% for 7680-bit operands. Furthermore, the operating latency is also outperformed with high clock frequency for length-64 transform and above.

Index Terms—Montgomery modular multiplication, number-theoretic weighted transform, fast Fourier transform (FFT), field-programmable gate array (FPGA).

1 Introduction

The subject of the paper is hardware implementations of the RSA algorithm [1] with larger than 1024-bit modulus length. In particular, our objective is to create implementations that achieve high area-time efficiency, rather than creating very low area or ultra high speed implementations at the high cost of the other. The RSA algorithm, being the very first public-key encryption and digital signature algorithm since 1978, is ubiquitously deployed and used, from smart cards to cell phones and SSL boxes. Its security depends on the difficulty of factoring a modulus n to find its two prime factors p and q. The security is increased by selecting higher modulus, however at the expense of large circuit size or slow operational speed. The very first implementations of the RSA algorithm [2] in early 1980s assumed 512-bit modulus (and thus, two 256-bit primes) would be sufficient, but within a decade, advances in factorization methods increased the modulus length to 1024 bits. This has been the case for almost 2 decades, but now, as recently as 2010s, the security of 1024-bit was questioned. Many implementations now use 2048-bit modulus, while the National Institute of Standard and Technology (NIST) recommended [3] 3072-bit or 4096-bit modulus size for the near future in order to maintain RSA secure. Needless to say, larger key sizes lead to longer processing time and more hardware resource when computing, due to the fact the RSA computation requires the modular exponentiation ($x^m \bmod N$), which is computed by repeated modular multiplications. Therefore, the performance of modular multiplication has a direct impact on the efficiency of RSA computation, and therefore, high performance modular multipliers supporting 3072-bit or higher operand size are required.

Montgomery modular multiplication (MMM) is an efficient method to compute modular multiplication [4]. In MMM algorithm, the time-consuming trial division is replaced by multiplications and reductions modulo R, where the reductions are trivial by selecting R to be a power of 2. Due to this fact, integer multiplication has been extensively studied in order to improve MMM. Existing multiplication methods can be classified into two groups. Methods of the first group are performed only in time domain, including the schoolbook method, the Karatsuba method [5], and the Toom-Cook method [6]. Methods of the second group are performed in both time and spectral domains, including the Schönhage-Strassen algorithm (SSA) [7], the Fürer's method [8], and the modified Fürer's method [9], [10]. Since the fast Fourier transform (FFT) based algorithm is applied to the second group, a lower asymptotic complexity can be achieved compared to the methods in the first ones. There are many hardware implementations of these multiplication methods: the schoolbook method [11], [12], [13], [14], the Karatsuba method [15], [16], and SSA [17], [18], [19].

For large operand size of MMM (3072-bit and above), the state-of-the-art architecture is proposed by [18], which improves the FFT-based Montgomery product reduction (FMPR) algorithm [17]. In [18], the multiplication and addi-

- *W. Dai, D. Chen and R. Cheung are with the Department of Electronic Engineering, City University of Hong Kong, China. E-mail: {w.dai, donald.chen}@my.cityu.edu.hk, cccheung@ieee.org*
- *Ç. K. Koç is with the Department of Computer Science, Universty of California Santa Barbara, Santa Barbara, CA, USA. E-mail: koc@cs.ucsb.edu*

Corresponding author: D. Chen.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2601334, IEEE Transactions on Computers

tion steps of MMM are performed in spectral domain, and the time-spectral domain transforms are supported by FFT. As a result, better performance is achieved by the design of [18] compared to the non-FFT-based designs (i.e. [11] and [14]). This indicates the performance of MMM can be further improved by applying the FFT method. However, the zero-padding operation is considered to be the bottleneck of the complexity reduction in FFT, since it causes redundant operations and enlarges the transform length.

Several variants of FFT method have been proposed aiming at the avoidance zero-padding. Phatak and Goff's method performs the Montgomery modular reduction by one linear convolution and one cyclic convolution [20]. Their method could partly avoid the zero-padding issue, since only cyclic convolution can be computed without zero-padding. Later, Saldamlı and Koç proposed a digit-serial algorithm, where all steps are performed in spectral domain [21]. Since it requires no transform during the computation, zero-padding is not involved. As a trade-off, performing modular reduction in spectral domain is more complicated and costs more hardware recourse. Besides, the serial algorithm is not suitable for massive parallel computation. McLaughlin provided a better solution in [22], where a new framework with a modified version of MMM algorithm is proposed. For fixed parameters, McLaughlin's algorithm is also suitable for the FFT method, moreover, by applying cyclic and nega-cyclic convolutions directly to the modular multiplication steps, the zero-padding operation is avoided.

In this work, we propose an FFT-based MMM algorithm under McLaughlin's framework, where the time-spectral domain transform without zero-padding is the repeated basic operation. The proposed algorithm is named as FMLM^3. Hardware realization of the FMLM^3 is targeting on high area-time efficiency, so that both hardware resource cost and cycle requirement are analysed based on different parameter sets. The major contributions of this work are summarized as:

- The modular multiplication steps of FMLM^3 are computed by FFT method directly without zero-padding, so that a lower complexity is achieved;
- A modified version of the FMLM^3 is proposed, which further reduces the number of domain transforms from 7 to 5;
- A general parameter set selection method is proposed for given operand size, both Fermat and Pseudo Fermat numbers are applied to support efficient FFT computation;
- Pipelined architectures with single and double butterfly structures are designed and implemented in order to explore the relation between cycle requirement and number of butterfly structures.

The rest part of this paper is organized as following. Section 2 provides the necessary mathematical backgrounds related to this work. Section 3 provides the detailed FMLM^3 steps and the parameter specifications. Section 4 presents arithmetical improvements of the FMLM^3, and a general parameter set selection method. Pipelined architectures of the FMLM^3 are presented in Section 5, where architectures with single and double butterfly structures are implemented. In Section 6, the implementation results of the proposed FMLM^3 are obtained and compared with other related works. In the last section, Section 7, remarks are provided.

TABLE 1
Notation List.

Notation	Description
$T(x)$	Cyclic transform of x
$T'(x)$	Nega-cyclic transform of x
CC	Cyclic convolution
NCC	Nega-cyclic convolution
CT	Cyclic transform
NCT	Nega-cyclic transform

Notation	Definition	Description
N		Modulus of $\mathbb{Z}_N$
l		Max. operands size
u		Bit length of each digit
v		Parameter to generate P
B	2^u	Radix
P	2^v	Transform length / number of digits
R	B^P-1	Modulus of FMLM^3
Q'	B^P+1	Modulus of FMLM^3
c	$\lceil\frac{v+2u+1}{P}\rceil$ or $\lceil\frac{2v+3u}{P}\rceil$ or $\frac{1}{2}$	Parameter to generate $\mathbb{Z}_M$
M	$2^{cP}+1$	Ring size of NTT
ω	2^{2c}	Primitive Pth root of unity
A	2^c	Primitive Pth root of -1

2 Preliminaries

This section introduces the mathematical background of FFT-based modular multiplication. For the ease of reference, the parameters in this paper and their definitions are listed in Table 1.

2.1 Modular Multiplication Using Convolutions

For an arbitrary non-negative integer x, we represent the radix-B format of x as:

$$x = \sum_{i=0}^{P-1} x_i B^i, \tag{1}$$

where B is a positive integer, $0 \leqslant x_i < B$ with $i = 0, 1, \ldots, P-1$ are called the digits of x, and P is the number of digits. The collection of digits x_i is denoted as $\{x_i\}$.

With the similar relation between $\{y_j\}$ and y, the multiplication $z = xy$ is equivalent to a length-$2P$ cyclic convolution, and the components z_n of z can be obtained by:

$$z_n = \sum_{i+j=n \pmod{2P}} x_i y_j, \tag{2}$$

where $n = 0, 1, \ldots, 2P-1$. $\{x_i\}$ and $\{y_j\}$ are padded with P zeros, such that $x_i = y_j = 0$ for all $P \leqslant i, j \leqslant 2P-1$. This method is applied by [17] and [18], where each multiplication step requires $4P^2$ digit-level multiplications.

Crandall and Fagin [23] discovered that for some fixed values of p, one could use length-P convolutions to compute $z = xy \bmod p$ without zero-padding, and therefore avoids the redundant operations.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2601334, IEEE Transactions on Computers

When $p = 2^l - 1$, let $B = 2^u$ and ensure $u|l$, so that $l = u \cdot P$ and $2^l \equiv B^P \equiv 1 \pmod{p}$. Thus, $z = xy \bmod p$ is obtained by:

$$\begin{aligned} z \bmod p &= \sum_{n=0}^{P-1}\Big(\sum_{i+j=n} x_i y_j\Big)B^n + \sum_{n=P}^{2P-1}\Big(\sum_{i+j=n} x_i y_j\Big)B^n \\ &= \sum_{n=0}^{P-1}\Big(\sum_{i+j=n \,(\bmod\, P)} x_i y_j\Big)B^n \\ &= \sum_{n=0}^{P-1}(x *_P y)_n B^n, \end{aligned} \tag{3}$$

where $i, j = 0, 1, \ldots, P-1$, and $(x *_P y)$ denotes a length-P cyclic convolution (CC) of $\{x_i\}$ and $\{y_j\}$. It can be observed from equation (3) that the zero-padding is avoided when computing $z = xy \bmod p$ compared to $z = xy$.

When $p = 2^l + 1$, we adopt the same conditions of l, B and P. The computation of $z = xy \bmod p$ is then the nega-cyclic convolution (NCC) $(x \bullet_P y)$ of $\{x_i\}$ and $\{y_j\}$ [23]. Thus, z is obtained by:

$$\begin{aligned} z \bmod p &= \sum_{n=0}^{P-1}\Big(\sum_{i+j=n} x_i y_j\Big)B^n + \sum_{n=P}^{2P-1}\Big(\sum_{i+j=n} x_i y_j\Big)B^n \\ &= \sum_{n=0}^{P-1}\Big(\sum_{i+j=n \,(\bmod\, P)} (-1)^s x_i y_j\Big)B^n \\ &= \sum_{n=0}^{P-1}(x \bullet_P y)_n B^n, \end{aligned} \tag{4}$$

where $s = \lfloor \frac{i+j}{P} \rfloor$. Again, the zero-padding in this case is avoided.

2.2 Number-Theoretic Weighted Transform

The number theoretic transform (NTT) provides a special domain, called spectral domain, in which a CC can be computed by component-wise multiplications [24]. Similarly, a NCC can also be computed component-wisely when applying the number-theoretic weighted transform (NWT) [23]. The forward and inverse NWT over ring $\mathbb{Z}_M$ are defined as:

$$\begin{aligned} X_n &= \sum_{k=0}^{P-1} x_k A^k \omega^{-kn} \pmod{M}, \\ x_n &= (A^n P)^{-1} \sum_{k=0}^{P-1} X_k \omega^{kn} \pmod{M}, \end{aligned} \tag{5}$$

where $n = 0, 1, \ldots, P-1$, $\{A^k\}$ is a non-zero weight sequence, and ω is the primitive P-th root of unity in $\mathbb{Z}_M$. The straightforward relationship between NWT and NTT is expressed as follows:

$$\begin{aligned} \{X_n\} &= \mathrm{NWT}(\{x_k\}) = \mathrm{NTT}(\{x_k A^k\}), \\ \{x_n\} &= \mathrm{NWT}^{-1}(\{X_k\}) = \{A^{-n}\}\mathrm{NTT}^{-1}(\{X_k\}). \end{aligned} \tag{6}$$

Obviously, $A = 1$ can be taken for the cyclic case (NWT is equivalent to NTT in this case), or A is a primitive P-th root of -1 ($A^P \equiv -1 \bmod M$) [23] for the nega-cyclic case. We rename the corresponding NWT as cyclic transform (CT) and nega-cyclic transform (NCT), respectively. Note that $\mathbb{Z}_M$ supports a length-P CC [24], or NCC [23] if and only if $P|(M_i - 1)$, where M_i is the prime factor of M.

Algorithm 1 McLaughlin's Montgomery modular multiplication [22]

Input: Ensure $R = 2^l - 1 > N$ and $\gcd(R, N) = 1$; for arbitrary $x', y' \in \mathbb{Z}_N$, let $x \equiv x'R \bmod N$, $y \equiv y'R \bmod N$; pre-compute R^{-1} and N', which satisfy $RR^{-1} - NN' = 1$, $0 < N' < R$, and $0 < R^{-1} < N$; select $Q' = 2^l + 1$ and $Q = 2Q'$, where $\gcd(R, Q) = 1$

Output: $t \equiv t'R = x'y'R = xyR^{-1} \pmod{N}$

1: $m = xyN' \pmod{R}$
2: $S = (xy + mN) \pmod{Q'}$
3: $w = -S \pmod{Q'}$
4: If $2|w$, then $s = w/2$, else $s = (w + Q')/2$
5: If $(xy + mN) \equiv s \pmod{2}$, then $t = s$, else $t = s + Q'$
6: If $t < N$, **return** t, else **return** $t - N$

The selection of transform length P as a power of 2 enables the radix-2 FFT algorithm [25], which leads to an acceleration of NWT and a reduction of digit-level multiplications from P^2 to $P \log_2 P$. The j-th stage FFT computation is shown in (7) for CT, (8) for NCT, (9) for both CT^{-1} and NCT^{-1}. Note that $J = 2^{\log_2 P - 1 - j}$, where $0 \leqslant n \leqslant \frac{P}{2} - 1$, and $0 \leq j \leqslant \log_2 P - 1$.

$$\begin{cases} x_n^{(j+1)} = x_{2n}^{(j)} + x_{2n+1}^{(j)} w^{-\lfloor n/J \rfloor \cdot J} \\ x_{n+P/2}^{(j+1)} = x_{2n}^{(j)} - x_{2n+1}^{(j)} w^{-\lfloor n/J \rfloor \cdot J} \end{cases} \tag{7}$$

$$\begin{cases} x_n^{(j+1)} = x_{2n}^{(j)} + x_{2n+1}^{(j)} w^{-\lfloor n/J \rfloor \cdot J + J/2} \\ x_{n+P/2}^{(j+1)} = x_{2n}^{(j)} - x_{2n+1}^{(j)} w^{-\lfloor n/J \rfloor \cdot J + J/2} \end{cases} \tag{8}$$

$$\begin{cases} X_n^{(j+1)} = X_{2n}^{(j)} + X_{2n+1}^{(j)} w^{\lfloor n/J \rfloor \cdot J} \\ X_{n+P/2}^{(j+1)} = X_{2n}^{(j)} - X_{2n+1}^{(j)} w^{\lfloor n/J \rfloor \cdot J} \end{cases} \tag{9}$$

Let $\mathrm{FFT}(\{x_i\})$ and $\mathrm{FFT}^{-1}(\{X_n\})$ denote the forward and inverse FFT of $\{x_i\}$ and $\{X_n\}$, respectively, and let $\odot$ denote the component-wise multiplication, $xy \bmod p$ can be computed efficiently with lower digit-level complexity:

$$\begin{aligned} xy \bmod p =& (x *_P y) \text{ or } (x \bullet_P y) \\ =& \mathrm{FFT}^{-1}[\mathrm{FFT}(\{x_i\}) \odot \mathrm{FFT}(\{y_j\})]. \end{aligned} \tag{10}$$

Computing equation (10) requires $3P \log_2 P + P$ digit-level multiplications, while equations (3) and (4) require P^2 and equation (2) requires $4P^2$.

2.3 McLaughlin's Montgomery Modular Multiplication

Instead of obtaining the modular product $xy \pmod{N}$ directly, Montgomery [4] introduces an extra fixed integer R, and computes $xyR^{-1} \pmod{N}$. Therefore, the MMM can efficiently avoid the time-consuming trial division.

McLaughlin proposed a modified version of MMM [22], the detailed computational steps are provided in Algorithm 1. Unlike the original version where R equals to a power of 2, the modified version redefines $R = 2^l - 1$ with an additional modulus $Q' = 2^l + 1$. McLaughlin's algorithm has a faster estimated running time compared to the original one. Moreover, for fixed modulus R and Q', the CT and NCT can be applied to the modular multiplication steps

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2601334, IEEE Transactions on Computers

Algorithm 2 Proposed FFT-based Montgomery modular multiplication under McLaughlin's framework (FMLM^3)

Input: Maximum operand size l, satisfy $2^l > N$; R, R^{-1}, N and N' satisfy $RR^{-1} - NN' = 1$, ensure $R = 2^l - 1 > N$, $\gcd(R, N) = 1$, $0 < R^{-1} < N$, $0 < N' < R$; precompute $T'(N)$ and $T(N')$; for arbitrary $x, y \in \mathbb{Z}_N$, precompute $T(x)$, $T'(x)$, $T(y)$, and $T'(y)$; select $Q' = R + 2 = 2^l + 1$

Output: $T(t), T'(t)$, where $t = xyR^{-1} \pmod N$

Compute $m = xyN' \bmod R$:

1: $T(g) = T(x) \odot T(N') \bmod M$
2: $g = \text{CT}^{-1}(T(g))$
3: $z_0 = g \bmod R$
4: $T(z_0) = \text{CT}(z_0)$
5: $T(z_1) = T(y) \odot T(z_0) \bmod M$
6: $z_1 = \text{CT}^{-1}(T(z_1))$
7: $m = z_1 \bmod R$
8: $T'(m) = \text{NCT}(m)$

Compute $S = xy + mN \bmod Q'$:

9: $T'(z_2) = T'(x) \odot T'(y) \bmod M$
10: $T'(z_3) = T'(m) \odot T'(N) \bmod M$
11: $T'(z_4') = T'(z_2) + T'(z_3)$
12: $z_4' = \text{NCT}^{-1}(T'(z_1))$
13: Restrict z_4' using equation (12) to obtain z_4
14: $S = z_4 \bmod Q'$
15: If $2|S$, $s = \frac{2Q'-S}{2} \bmod Q'$, else $s = \frac{Q'-S}{2}$
16: If $xy + mN \equiv s \pmod 2$, $t = s$, else $t = s + Q'$
17: If $t \geqslant N$, $t = t - N$
18: **Return** $T(t) = \text{CT}(t)$
19: **Return** $T'(t) = \text{NCT}(t)$

efficiently without zero-padding (cf. Section 2.2). Due to this fact, FFT method is also suitable for McLaughlin's algorithm.

3 FFT-based Montgomery Modular Multiplication under McLaughlin's Framework

In this section, an FFT-based Montgomery modular multiplication under McLaughlin's Framework [22] (FMLM^3) is proposed, parameter specification of the algorithm is also introduced.

3.1 The Proposed Algorithm of FMLM^3

The FFT method can be applied to Algorithm 1 to perform efficient multiplications modulo R and Q', the FFT-based algorithm (FMLM^3) is provided as shown in Algorithm 2.

The computation of FMLM^3 starts from either time or spectral domain, which depends on the type of input data and the output should be consistent with the input. Algorithm 2 starts the computation from spectral domain, thus two more steps are required to obtain $T(t)$ and $T'(t)$.

In Step 12 of Algorithm 2, the results of NCT^{-1} should be equivalent to the components of NCC, which can be extracted from equation (4):

$$\begin{aligned} z_n &= \sum_{i+j=n} x_i y_j - \sum_{i+j=n+P} x_i y_j \\ &= \sum_{i=0}^{n} x_i y_{n-i} - \sum_{i=n+1}^{P-1} x_i y_{n+P-1}, \end{aligned} \tag{11}$$

TABLE 2
Complexity comparison between the proposed FMLM^3 and other modular multiplications.

Algorithm	Barrett	Montgomery		
Design	[20]*	[26]	[18]	FMLM^3
Method	FFT	RNS	FFT	FFT
Digit-level complexity	$10P\log_2 P + 13P$	$18P^2 + 15P$	$12P\log_2 P + 18P$	$7P\log_2 P + 4P$

* Estimated digit-level complexity.

where z_n is the n-th component of NCC, and $i, j, n = 0, 1, \ldots, P-1$. Clearly, for $x_i, y_j \in [0, B)$, each z_n has a lower bound $-(P-1-n)(B-1)^2 \leqslant 0$ and an upper bound $(n+1)(B-1)^2 > 0$. The lower bound indicates that a negative component may be obtained by NCC. However, all components of z_4' (Step 12) are within the range of $[0, M)$, since NCT^{-1} is performed in ring $\mathbb{Z}_M$. Thus, the components of z_4' must be restricted to equation (12) by subtracting M before obtaining z_4. This restriction guarantees a correct result of NCC using the FFT method.

$$-2(P-1-n)(B-1)^2 \leqslant z_n \leqslant 2(n+1)(B-1)^2. \tag{12}$$

Due to the fact that the sum of two NCCs are computed, where the addition takes place in the Step 11, therefore, both the lower and upper bounds in equation (12) are doubled.

The FMLM^3 is able to use length-P transforms, while the algorithm of [18] requires length-$2P$ transforms because of the zero-padding. This is considered to be the major advantage of FMLM^3. When computing modular exponentiation $x^k \bmod p$ using the FMLM^3, $T'(N)$ and $T(N')$ can be reused. Besides, as mentioned in [22], the result of $xN' \bmod R$ ($T(z_0)$ in Step 4 of Algorithm 2) can also be reused, this reduces the number of transforms from 7 to 5, more specifically, Steps 1-4 can be saved during the modular exponentiation.

A complexity comparison is provided in Table 2 between the FMLM^3 and other modular multiplication methods. RNS refers to the residue number system, which is another divide-and-conquer approach to compute MMM. Since [20] only considers the reduction steps, the complexity of Barrett modular multiplication is estimated by assuming that the multiplication step requires $4P\log_2 P + 6P$ digit-level multiplications. It can be observed that the FMLM^3 has the lowest complexity among all compared algorithms.

3.2 Parameter Specifications

In order to perform fast computation of the FMLM^3, we choose $B = 2^u$ so that representing an integer in radix-B form is simply a bitwise partition; besides, we select $P = 2^v$ to enable the radix-2 FFT computation. Thus, the maximum supported operand size is defined as $l = \log_2 B^P = u \cdot 2^v$. By substituting l, R and Q' can be redefined as $R = B^P - 1$ and $Q' = B^P + 1$, respectively.

In order to avoid data overflow during the NWT computation, the ring size M must satisfies:

$$M > 2(B-1)^2 P. \tag{13}$$

In fact, ensuring $M > (B-1)^2 P$ already maintains the precision of modular multiplication. We add one extra bit

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2601334, IEEE Transactions on Computers

TABLE 3
Supported operand size of Fermat numbers F_v, when $v = 6, 7$.

Max. Operand size l	Ring size M	Transform length $P = 2^v$	Roots ω / A	Digit bit length u
1,792	$F_6 = 2^{64} + 1$	64	4 / 2	28
3,584	$F_6 = 2^{64} + 1$	128	2 / $\sqrt{2}$	28
7,680	$F_7 = 2^{128} + 1$	128	4 / 2	60
15,104	$F_7 = 2^{128} + 1$	256	2 / $\sqrt{2}$	59

TABLE 4
Examples of eligible parameter sets targeting $l = 2048$.

Set no.	$B = 2^u$	$P = 2^v$	c	M	w / A
1.	2^{128}	16	17	2^{272}+1	2^{34} / 2^{17}
2.	2^{64}	32	5	2^{160}+1	2^{10} / 2^5
3.	2^{32}	64	2	2^{128}+1	2^4 / 2^2
4.	2^{16}	128	0.5	2^{64} +1	2 / $2^{48}-2^{16}$
5.	2^8	256	0.5	2^{128}+1	2 / $2^{96}-2^{32}$
6.	2^4	512	0.5	2^{256}+1	2 / $2^{192}-2^{64}$

in (13) so that xy and mN (Step 11 of Algorithm 2) can be component-wisely added prior to the NCT^{-1}. This avoids the long carry chain when performing the addition in time domain. To apply fast reduction computation, modulus M should have low Hamming weight. Thus, we let M to be a Fermat number $F_v = 2^P + 1$, where $P = 2^v$.

According to [23], for a composite M, one can always define a length-2^{v+1-i} NWT with $\omega = 2^{2^i}$ and $A = 2^{2^{i-1}}$ over $\mathbb{Z}_M$, where $i = 0, 1, \ldots, v + 1$. Practically, A should satisfy two conditions:

- A is as small as possible, which results in a larger l;
- A has simple expressions so that multiply by A^k can be computed easily by shifts and additions.

Consequently, the smallest A is $\sqrt{2}$ when $i = 0$, where A has an expression of:

$$A \equiv 2^{2^{v-3}}(2^{2^{v-2}} - 1) \bmod 2^{2^v} + 1. \tag{14}$$

Following the previous parameter specifications, we select two Fermat numbers F_6 and F_7, and summarize the supported parameter sets in Table 3 as examples.

The Fermat numbers in Table 3 support the major key sizes of RSA (i.e. 1024, 2048, 3072, 4086 and 7680-bit) suggested by the NIST [3] and ECRYPT [27]. However, it is hard to find an appropriate F_v to support an l which is closed or equal to the suggested key size.

In order to narrow the gap between l and the suggested key size, pseudo Fermat numbers $F = 2^{c2^v} + 1$ ($c \geqslant 2$) are employed to define a length-2^{v+1-i} NWT [28]. Thus, we introduce an extra parameter c, and redefine $M = 2^{cP} + 1$, $\omega = 2^{2c}$ and $A = 2^c$, respectively. Based on (13), c should satisfy:

$$c = \frac{1}{2} \geqslant \frac{v + 2u + 1}{P} \quad \text{or} \quad c = \lceil \frac{v + 2u + 1}{P} \rceil. \tag{15}$$

The application of pseudo Fermat number and parameter c can provide us more flexible choices when generating the parameter sets. Table 4 provides eligible parameter sets targeting on 2048-bit key size. Note that the maximum operand size l is exactly 2048-bit without "wasting" any bit.

4 Optimizations of the FMLM^3

In this section, fast modular reduction algorithms are introduced and a modified version of the FMLM^3 is proposed. Based on these optimizations, an efficient parameter set selection method is then summarized.

4.1 Modulo R Reduction and Redundant Representation

In Step 2 of Algorithm 2, g is obtained by:

$$g = \sum_{i=0}^{P-1} g_i 2^{ui}, \tag{16}$$

where g_i is the i-th component of CT^{-1}. Since $g_i \leqslant (B - 1)^2 P$, the maximum bit length of g is $uP + u + v + 1 > l$. This implies g may be larger than R, an extra reduction is required to reduce g within the range of $[0, R)$ (Step 3). With $R = 2^{uP} - 1$, $g \bmod R$ can be operated in two steps. Firstly, we compute:

$$g' = g_H + g_L = \lfloor \frac{g}{2^{uP}} \rfloor + (g \bmod 2^{uP}). \tag{17}$$

Since g' equals to either z_0 or $z_0 + R$, where $z_0 \equiv g \bmod R$, a second step is required to correct the case when $g' = z_0 + R$ by subtracting R from g'. It can be observed that providing $z_0 + R$ in Step 3 is tolerable, since the remaining extra R can be reduced by the second modulo R reduction in Step 7. Therefore, Step 3 of Algorithm 2 can be computed by only one addition as shown in equation (17), and the correction step can be saved.

To perform fast computation of equation (17), we introduce the redundant representation. For radix-B representation of x, each x_i has u bits. If x_i maintains one extra bit precision ($u+1$ bits) on the same basis, then we call x is in its redundant representation. There are more than one radix-B redundant representation for each x. Taking $x = 871206$ and $B = 2^5$ as an example, the radix-B representation of x is $(x_0, x_1, x_2, x_3) = (6, 25, 18, 26)$, while the radix-$B$ redundant representation of x can be either $(38, 24, 50, 25)$ or $(6, 57, 49, 25)$.

When applying redundant representation to compute equation (17), we add the two operands digit-by-digit, and prevent the carry bit from propagating to the next digit-level addition. Thus, the sum of each two digits has maximumly $u + 1$ bits, the g' is then in its redundant form. Under such circumstances, M must satisfy $M > P(B - 1)(2B - 2) = 2P(B-1)^2$, which is same as the condition in (13). Therefore, no additional restriction of M is required when applying redundant representation.

This acceleration approach is only available for Step 3 of Algorithm 2. The second modulo R reduction in Step 7 is followed by a different modulus Q', so an accurate result of the reduction is required.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2601334, IEEE Transactions on Computers

Algorithm 3 Modified version of the proposed FMLM^3

Input: Requirement is the same as Algorithm 2
Output: $T(t), T'(t)$, where $t = xyR^{-1} \pmod N$
Compute $m = xyN' \bmod R$:
1: $T(z_0) = T(x) \odot T(N') \bmod M$
2: $T(z_1) = T(y) \odot T(z_0) \bmod M$
3: $z_1 = \mathrm{CT}^{-1}(T(z_1))$
4: $m = z_1 \bmod R$
5: Step 8-19 of Algorithm 2

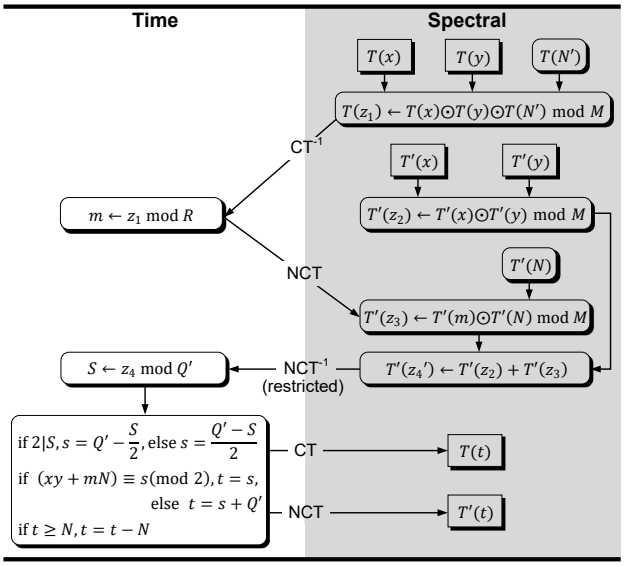

Fig. 1. Data flow of the modified FMLM^3 (Algorithm 3), the right-angled blocks represent input/output data, $T(N')$ and $T'(N)$ are pre-computed.

4.2 Modulo M Reduction

Modulo M reduction is one of the basic operations in the FMLM^3. As introduced in [29], modulo M reduction requires two steps: first, divide operand x into digits on radix-cP basis and computed:

$$x \bmod M = \sum_{i=0} (-1)^i x_i. \tag{18}$$

Then, correct the result to range $[0, M)$. In order to avoid the correction step, signed operations are applied (cf. Section 5.5 of [18]), which will increase the data width to $(cP + 2)$-bit.

4.3 Reduce the Number of Transforms in the FMLM^3 Algorithm

In Algorithm 2, the computation of $T'(m)$ requires 4 NWTs, since $T(x)$, $T(y)$ and $T(N')$ are multiplied sequentially. This number can be reduced to 2 by multiplying the three operands "all at once", as shown in Algorithm 3. Fig. 1 provides the data flow of the modified FMLM^3.

Compared with Algorithm 2, the modified version reduces the number of NWTs from 7 to 5. Benefit from this improvement, the Algorithm 3 has a lower complexity and a simpler data flow. As a trade-off, a larger dynamic range of M is required:

$$M > P^2(B-1)^3, \tag{19}$$

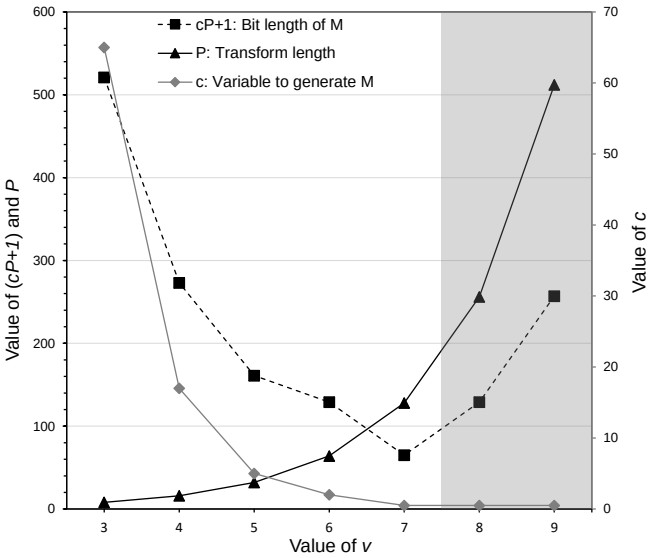

Fig. 2. Relationship between transform length P, bit length of M which equals $cP+1$, and c in terms of v. The targeted operand size is $l = 2048$. The left vertical axis refers to $cP + 1$ and P, the right one refers to c.

accordingly

$$c = \frac{1}{2} \geqslant \frac{2v + 3u}{P} \quad \text{or} \quad c = \lceil \frac{2v + 3u}{P} \rceil. \tag{20}$$

Compared with (13), for the same values of u and v, a larger M may be obtained in order to satisfy new restriction (19). Taking the parameter sets in Table 4 as examples: when applying Set 1 to Algorithm 3, c and M will be enlarged to 25 and $2^{400} + 1$, respectively. However, Sets 3-6 can be applied directly to Algorithm 3 with no changes. This implies a "free" number reduction of NWTs from 7 to 5.

4.4 Efficient Parameter Set Selection

The efficiency of FMLM^3 is impacted by the parameters, especially by the transform length P and ring size M. A larger P implies more digit-level multiplications; a larger M implies larger operands of each digit-level operation. Fig. 2 is depicted in order to explore the relation between the parameters in Table 4 for $l = 2048$.

In Fig. 2, the value of P increases along with the increase of v. While M decreases until $c = 0.5$. However, after $c = 0.5$, both M and P go up with the increase of v, thus, the complexities of the corresponding parameter sets are always higher than the previous ones. For example, when $v = 6$ (Set 3 in Table 4) and $v = 8$ (Set 5), a same M is required, but the transform length of Set 3 ($P = 64$) is shorter than that of Set 5 ($P = 256$). Clearly, Set 3 requires less digit-level multiplications, and therefore, Set 3 is more efficient than Set 5. In summary, the parameter sets in the shadowed area in Fig. 2 are all considered to be inefficient. However, hardware realizations are required to further evaluate the efficiency of rest parameter sets. We generalize the analysis above and proposed a parameter set selection method in Algorithm 4 to avoid the inefficient sets as just discussed.

5 Pipelined Architecture of FMLM^3

The FMLM^3 includes two different modulus $R = 2^{2^v} - 1$ and $Q' = 2^{2^v} + 1$ and performs fast NWT without zero-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2601334, IEEE Transactions on Computers

Algorithm 4 Efficient parameter set selection method

Input: Targeted operand size l

Output: Parameter sets $\{l, v, u, P, B, c, M, w, A\}$

1: Let $P = 2^v$, v is positive integer, traversal eligible P from 2 to l, when $P = 1$, the FMLM^3 is equivalent to schoolbook multiplication.
2: Calculate digit size $u = \lceil \frac{l}{P} \rceil$ and radix $B = 2^u$.
3: Generate $R = B^P - 1$ and $Q' = B^P - 1$.
4: If $\frac{v+2u+1}{P} \leqslant \frac{1}{2}$, $c = \frac{1}{2}$, else $c = \lceil \frac{v+2u+1}{P} \rceil$; if $\lceil \frac{2v+3u}{P} \rceil = \lceil \frac{v+2u+1}{P} \rceil$, apply Algorithm 3, else apply Algorithm 2.
5: Let $M = 2^{cP} + 1$, $w = 2^{2c}$ and $A = 2^c$ are primitive P-th root of unity and (-1), respectively.
6: Update $l = u \cdot 2^v$, l is the practical maximum operand size, record $\{l, v, u, P, B, c, M, w, A\}$ as an eligible parameter set.
7: If $c = \frac{1}{2}$, stop selection, else repeat 1-6.

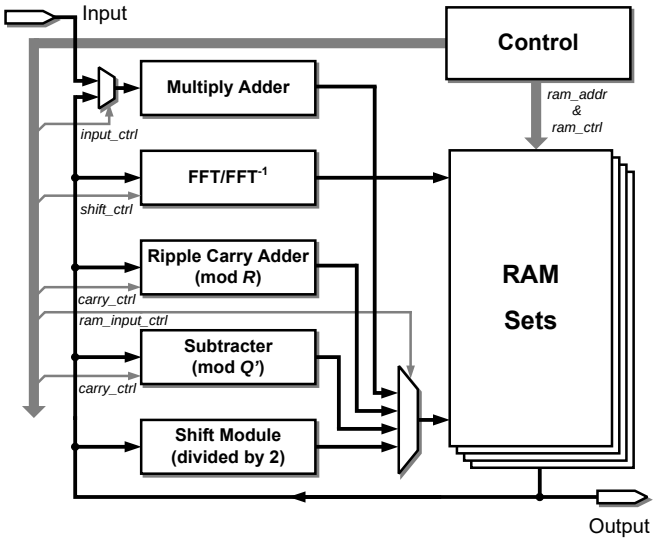

Fig. 3. Top level architecture of the proposed FMLM^3. The *Control* unit generates one control code at each clock cycle, control code includes all the necessary control signals, for instance: ram_ctrl signals control the behavior of each RAM; $shift\_ctrl$ signals control bit-wise shifting during the transforms.

padding. This algorithm has a more complicated data flow compared to the repeatable structure proposed in [18]. Also, more operation units are required to deal with the modular reductions and the conditional selections. The top-level architecture of the FMLM^3 is proposed in Fig. 3. Operations of the FMLM^3, from the top-level view, are computed sequentially, while pipelined architectures are designed inside each unit. The forward and inverse NWTs are performed in the *FFT/FFT*$^{-1}$ unit. Component-wise multiplication and addition are performed in the *Multiply Adder* unit. The *Ripple Carry Adder (RCA)*, the *Subtracter* and the *Shift Module* units are responsible for the time domain operations, such as modulo R and Q' reductions, conditional selections, etc. A *Control* unit is designed to generate all the control signals of the entire system. The *RAM* unit, which consists of several RAM sets, stores the pre-computed data, the intermediate results, and the final modular product.

5.1 FFT/FFT^{-1} Unit

The architecture of our design is targeted on high clock frequency while maintaining a small resource cost. The pipelined butterfly structure (BFS) proposed by [18] is adopt in our *FFT/FFT*$^{-1}$ unit to achieve this goal. Instead of the in-place FFT, the constant geometry FFT is applied to the FFT computation (cf. Fig. 6 of [18]). Compared with the in-place FFT, constant geometry FFT has a same connection network between every adjacent stages, which results in a simpler read-and-write control. In order to explored the trade-off between hardware resources and latency, both architectures with 1 and 2 BFSs are built. Fig. 4 provides the pipelined architecture of the *FFT/FFT*$^{-1}$ unit which integrates 2 BFSs.

Since the FMLM^3 employs two types of NWTs, CT and NCT, a more complex architecture is designed compared to [18]. Apart from the two BFSs, a Channel Switcher (CS), which consists of eight 2-to-1 MUX arrays is designed to ensure the intermediate digits can be written into the correct RAM location. Moreover, two Final Stage Operators (FSO A and B) are designed to compute the final stage of the inverse NWTs (cf. Fig. 4). Note that during the final stage computation, the results of channel 0 and 1 are forwarded to the FSOs first, concurrently, the results of channel 2 and 3 are stored into the buffer and will be operated after completing the computations of channel 0 and 1.

The *FFT/FFT*$^{-1}$ unit is designed with six inputs, four inputs forward the digits into BFSs for the FFT computation, the other two inputs forward the pre-computed upper bound constraints into FSOs to restrict the results of NCT^{-1} before entering the final accumulation. Before applying the constraints (12) of NCT^{-1}, the unconstrained digits are non-negative and equal to either x_n (constraint met) or $x_n + M$ (constraint not met). Since the lower bounds of (12) are non-positive integers, while $x_n \in [0, M)$, we only need to check the upper bounds and correct the $x_n + M$ cases by subtracting M.

An accumulator is designed to recombine the results of CT^{-1} or NCT^{-1}. Two adjacent digits are added at each cycle and the results are generated in two pipeline stages. In the first stage, it computes:

$$R_0 = x_{i+1}2^u + x_i, \tag{21}$$

where x_i and x_{i+1} denotes the two input digits. In the second stage, R_0 is added to the accumulation register and two radix-B digits are generated:

$$\begin{aligned} r_{i+1} &= R_0 + \lfloor \frac{r_i}{2^{2u}} \rfloor, \\ X_{i+1}2^u + X_i &= r_i \bmod 2^{2u}, \end{aligned} \tag{22}$$

where X_i, X_{i+1} denote the two output digits, r_i denotes the data stored in accumulation register ($r_0 = 0$). It worth to note that in our design, the multiplication, division and reduction in both (21) and (22) are performed efficiently by shift or bitwise partition.

Shift operators are designed to compute the times power-of-2 operations during the NWT computation. Control signals $shift\_ctrl0$ and $shift\_ctrl1$ transmit the twiddle factors (the number of ω) to handle the shift operation. The j-th stage shift bits are obtained according to equations (7), (8), and (9). Since the inverse NWTs are scaled by P^{-1} or $(A^nP)^{-1}$ (when $n = 0$, $(A^nP)^{-1} = P^{-1}$), one more

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2601334, IEEE Transactions on Computers

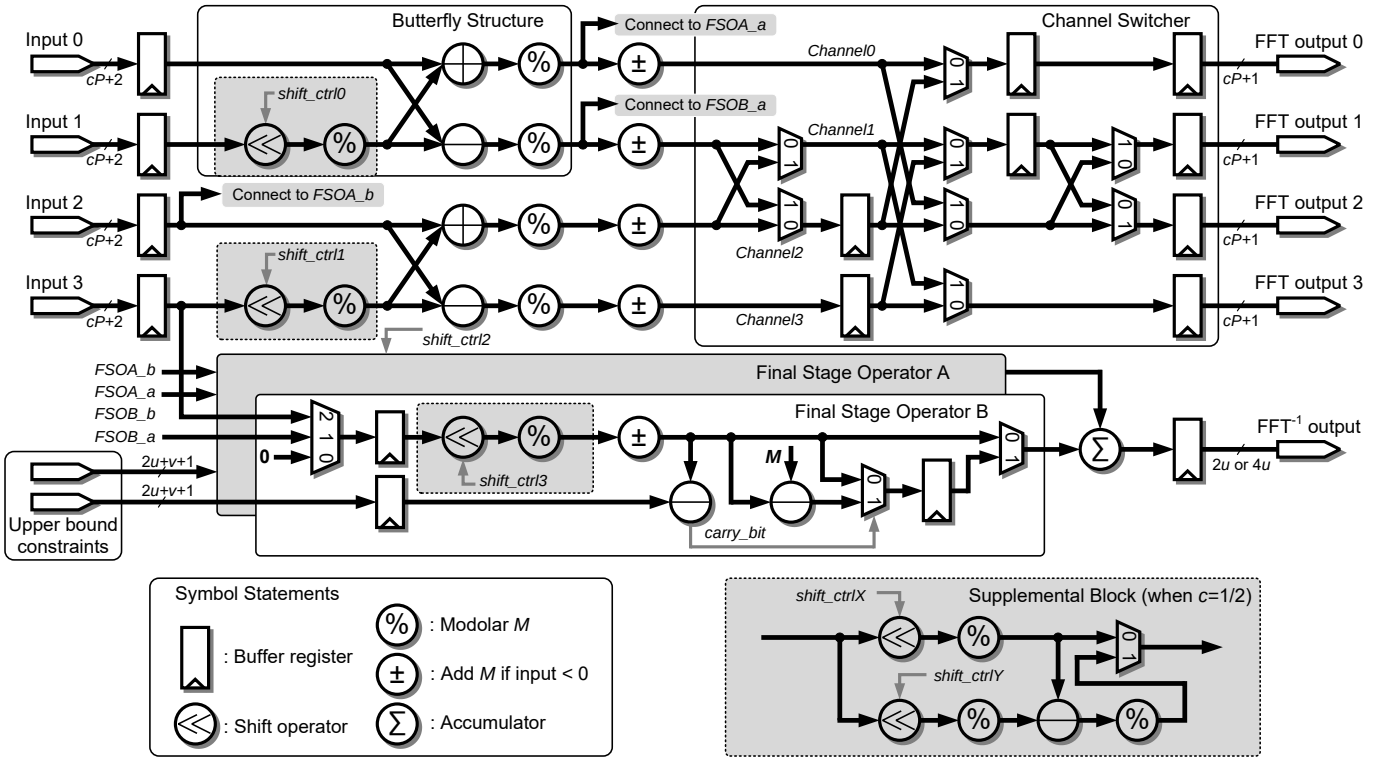

Fig. 4. Pipelined architecture of *FFT/FFT*$^{-1}$ with 2 butterfly structures. $shift\_ctrl\#$ signals are responsible for the multiplication of powers of 2 during the transform. The three dashed blocks are replaced by the Supplemental Block when $c = 1/2$. A Channel Switcher is used to reorder the output digits, and ensure operations of the following stage can be computed correctly. Switch control signals of the MUXs and registers following the operators are omitted. The Final Stage Operators A and B are responsible for the last stage of CT^{-1} and NCT^{-1}.

shift operator is integrated in each of the FSOs (controlled by $shift\_ctrl2$ and $shift\_ctrl3$, respectively). When $c \neq \frac{1}{2}$, $A = 2^c$ is a rational number, and ω has integer powers in equations (7), (8), and (9). NWTs can be computed by the *FFT/FFT*$^{-1}$ unit without the Supplemental Block (cf. Fig 4). When $c = \frac{1}{2}$, $A \equiv \sqrt{2} \equiv 2^{3 \cdot 2^{v-3}} - 2^{2^{v-3}} \bmod M$, therefore, one subtraction, two shifts and three modulo M reductions are required to multiply A.

For the case $c = \frac{1}{2}$, the computation of NCT and NCT^{-1} require more operations compared to CT and CT^{-1} due to the non-integer power of ω and the scale of the irrational number. Considering the NCT computation when $A = \sqrt{2}$, the twiddle factors are obtained by $\omega^{-\lfloor n/J \rfloor \cdot J + J/2}$ according to equation (8), where $J = 2^{v-1-j}$. This indicates the power of ω is not an integer only in the final NCT stage ($j = v-1$). The Supplemental Block will replace the dashed block in BFSs in this case (cf. Fig. 4). The MUX in the Supplemental Block will select the lower output during the final stage computation while selecting the upper one for the rest of the stages. By substituting $J = 1$ and $\omega = 2$, shift bits of the final stage is obtained by:

$$\omega^{-\lfloor n/J \rfloor \cdot J + J/2} = 2^{3 \cdot 2^{v-3} - n} - 2^{2^{v-3} - n} \bmod M. \quad (23)$$

Considering the NCT^{-1} computation when $A = \sqrt{2}$, since NCT^{-1} is scaled by $(A^n P)^{-1}$, the final stage of NCT^{-1} is divided into two cases. When n is even, $A^n = 2^{\frac{n}{2}}$ is always an integer, so $(A^n P)^{-1} \equiv 2^{2^v - v - \frac{n}{2}} \bmod M$ is simply a power of 2. However, when n is odd, computing $(A^n P)^{-1}$ is more complicated:

$$(A^n P)^{-1} \equiv (2^{3 \cdot 2^{v-3}-1} - 2^{2^{v-3}-1}) \cdot 2^{2^v - v - \frac{n-1}{2}} \bmod M. \quad (24)$$

Therefore, when $A = \sqrt{2}$, FSO A remains no change since n is always even in this channel, while the dashed part of FSO B will be replaced by the Supplemental Block.

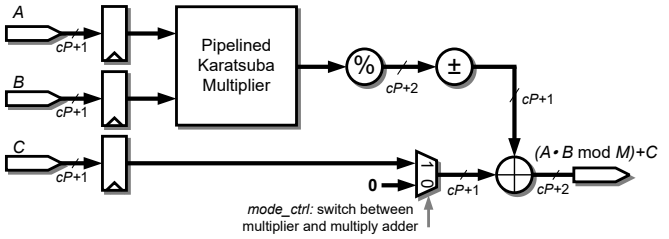

Fig. 5. Pipelined architecture of the *Multiply Adder* unit. There are registers followed after each operator, we omitted them for simplicity (refer to Fig. 4 for symbol descriptions).

5.2 Multiply Adder Unit

The *Multiply Adder* unit implements the component-wise multiplication and the addition of the FMLM^3. We chose the Karatsuba method [5] to realize the component-wise multiplication because it is efficient when the operand size is no larger than a few hundred bits [30]. The *Multiply Adder* unit is working in pipelined with three $(cP + 1)$-bit inputs (denote as A, B and C), and one output obtaining $(A \times B \bmod M) + C$, as shown in Fig. 5 (cf. Fig. 5 of [18]

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2601334, IEEE Transactions on Computers

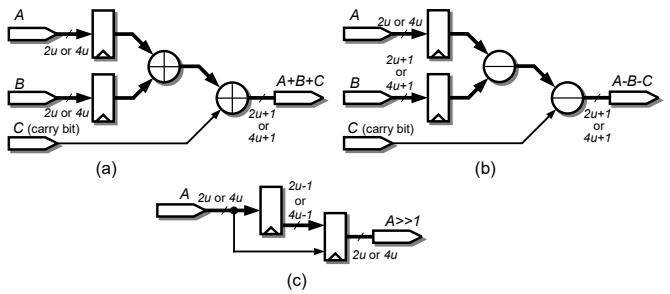

Fig. 6. Architectures of (a) the *Ripple Carry Adder* unit; (b) the *Subtracter* unit; (c) the *Shift Module* unit. The registers followed after each operator are omitted for simplicity.

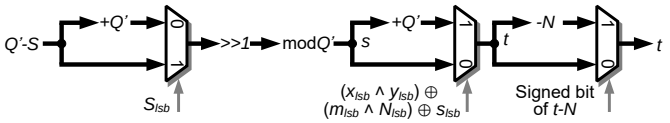

Fig. 7. Data flow of the conditional selections for Steps 15-17 in Algorithm 2. The subscript "lsb" denotes the least-significant-bit.

for detailed Karatsuba multiplier architecture). To enhance the performance of multiplication, the Karatsuba method is applied recursively and the operand size of the i-th recursion, denoted as $d^{(i)}$, is determined by:

$$d^{(i+1)} = \frac{d^{(i)} + 1}{2}, \tag{25}$$

where $d^{(0)} = cP + 1$. The product of A and B is reduced by modulus M. We also integrate a $(1 + cP)$-bit adder to perform the component-wise addition of Step 11 in Algorithm 2. The pipeline stage of the designed architecture is:

$$\Delta_{MA} = 4 + 4n + \Delta_{mul}, \tag{26}$$

where n denotes the number of recursions, Δ_{mul} denote the pipeline depth of the core multiplication unit.

5.3 Time Domain Operation Units

The *RCA*, *Subtracter* and *Shift Module* units are designed to implement the time domain operations (i.e. Steps 3, 7, 14-17 of Algorithm 2, and Step 4 of Algorithm 3). Note that the operand size of these operations can be as large as l bits, thus, long carry chains might be involved when computing them directly. Considering that the data width in *RAM* is $(cP + 1)$-bit to store the results of NWT; the data width is maintained either $cP + 2 \geqslant v + 2u + 3 > 2u$, or $cP + 2 > 4u$ during the FFT computation (cf. Table 4), we divide these large operands into $2u$ (or $4u$)-bit segments, and compute one segment per cycle to shorten the carry chain. Specifically, the *RCA* and *Subtractor* are constituted of two $2u$ (or $4u$)-bit cascaded adders and subtractors, respectively. Since each unit is designed to have 3 pipeline stages, outputting all $\frac{P}{2}$ (or $\frac{P}{4}$) results require $\frac{P}{2} + 3$ (or $\frac{P}{4} + 3$) cycles. In terms of the *Shift Module* unit, 2 pipeline stages with $2u$ (or $4u$)-bit input and output are designed, each division by 2 operation needs $\frac{P}{2} + 2$ (or $\frac{P}{4} + 2$) cycles. The detailed architectures are provided in Fig 6. This design guarantees that each addition/subtraction/shift can be computed as fast as possible without dragging down the clock frequency. For example, the time domain operations can be computed every $2u$-bit per cycle for Set 2 in Table 4 because $2u = 128 < cP + 2 = 162$. Moreover, the operations can be computed more efficient with $4u$-bit per cycle for Set 3 in Table 4, since $4u = 128 < cP + 2 = 130$.

5.3.1 Modulo R Reduction

As discussed in Section 4.1, the modulo R reduction includes two additions, where each addition requires $\frac{P}{2}$ (or $\frac{P}{4}$) cycles. Considering the read-and-write operation requires 3 cycles, performing Step 7 in Algorithm 2 and Step 4 in Algorithm 3 require $P + 3$ (or $\frac{P}{2} + 3$) cycles, respectively. When computing Step 3 in Algorithm 2, the second addition can be saved (cf. Section 4.1) and we only need to compute the first addition which is expressed in equation (17). Moreover, it is not necessary to go through all the segments of g_L when applying the redundant representation, since the bit length of g_H is $2u$ or less ($g_H = \lfloor \frac{g}{2^{uP}} \rfloor \leqslant 2^{u+v+1} - 1 \leqslant 2^{2u} - 1$). As a result, the first addition requires only 1 cycle, and performing Step 3 in Algorithm 2 requires 4 cycles when considering the read-and-write operation.

5.3.2 Modulo Q' Reduction

The computation of modulo Q' reduction includes two steps: one subtraction and one signed bit correction. The first step requires $\frac{P}{2} + 3$ (or $\frac{P}{4} + 3$) cycles, and the result is either a non-negative value S, or a negative value $S - Q'$. When applying the redundant representation, $S - Q'$ can be corrected by simply adding the signed bit 1 to the least-significant segment of $S - Q'$ and ignoring the signed bit (l-th bit). This is because:

$$S - Q' + 1 \bmod 2^l = S - Q' + 1 + 2^l = S. \tag{27}$$

Thus, the second step is computed efficiently in only 1 cycle by adding the signed bit. As a result, performing modulo Q' reduction requires $\frac{P}{2} + 4$ (or $\frac{P}{4} + 4$) cycles.

5.3.3 Conditional Selections

The data flow for the conditional selections in Steps 15-17 of Algorithm 2 is depicted in Fig. 7. For any addition/subtraction/shift operation, the data is loaded from the *RAM* and forwarded into the corresponding operation unit segment-by-segment, note that every segment here contains $2u$ or $4u$ bits. Then, after the computation, the results are generated and written back to the *RAM* segment-by-segment.

5.4 RAM Unit and Read/Write Control

The data storage requirement during the FMLM^3 computation is not trivial. For instance, the *FFT/FFT*$^{-1}$ unit generates $(cP^2 + P)$-bit intermediate results per stage; the *Multiply Adder* unit generates $(cP^2 + 2P)$-bit results for each computation. Moreover, $(3cP^2 + 3P + uP)$-bit storage space is required to store the pre-computed data. In order to well manage the input/output of the data and reduce the wiring workload, the *RAM* unit with block RAMs is built. The *RAM* unit is divided into three types:

- A dual port ROM is used to store the pre-computed data: $T(N')$, $T'(N)$, N, and the upper bound constraints of NCT^{-1};

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2601334, IEEE Transactions on Computers

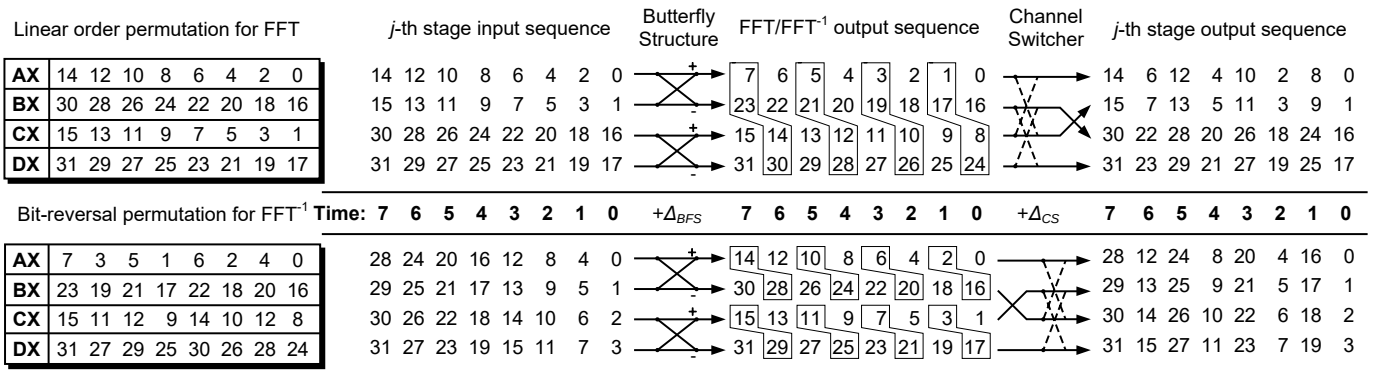

Fig. 8. Permutations and j-th stage data flow of a length-32 FFT and FFT^{-1}, respectively. Bit-reversal permutation is performed for FFT^{-1}, linear order permutation is performed for FFT. Every clock cycle, four digits are loaded from either RAM group 0 or 1. Δ_{BFS} and Δ_{CS} denote the pipeline depth of Butterfly Structure and Channel Switcher, respectively. Either dashed or solid path is selected each cycle to exchange the digits between Channel 0, 1 and Channel 2, 3.

• Eight simple dual port RAMs with $P/4$ read-and-write depth are used for the NWT computation. These RAMs are divided into two groups to support the data alternative storage mechanism [18];
• A true dual port RAM is used to store all the time domain operations. Also, it stores the data of $T'(z_2)$ (Step 9 of Algorithm 2), so that $T'(z_2)$ can be computed with $T'(m)$ (Step 8) in parallel.

Compared with the in-place FFT, the read/write control of the constant geometry FFT is much simpler. Specifically, in the j-th stage, if digits ($x_{2n}^{(j)}$, $x_{2n+1}^{(j)}$) are read from the *RAM* unit into the *FFT/FFT*$^{-1}$ unit (where $n = 0, 1, \ldots, P/2-1$), the results are always digits ($x_n^{(j+1)}$, $x_{n+P/2}^{(j+1)}$) of the next stage.

FFT and FFT^{-1} apply different permutations scheme. Permuting the digits of the FFT computation requires $\frac{P}{2}$ or $\frac{P}{4}$ cycles, this is because these digits are generated by the *RCA* or *Subtracter*, which 2 and 4 digits are output per cycle,respectively. In terms of FFT^{-1}, P cycles are required for permutation because the *Multiply Adder* generated one digit per cycle. Due to this fact, linear and bit-reversal order permutation are used for FFT and FFT^{-1}, respectively.

An example illustrating the j-th stage input/output orders and the permutations for 32-point FFT and FFT^{-1} are provided in Fig. 8. Let AX, BX, CX, and DX denote the RAMs of each group (X $= 0, 1$). As shown in the upper part of Fig. 8, in the j-th stage computation of FFT, (x_0, x_1) and (x_{16}, x_{17}) are loaded from the *FFT/FFT*$^{-1}$ unit at 0-th clock cycle. After Δ_{BFS} cycles, results (x_0, x_{16}) and (x_8, x_{24}) are obtained. Since x_8 and x_{24} belong to RAMs AX and BX, respectively, a CS is implemented to ensure the digits can be written into the correct RAMs. During the channel switch operation, we first switch channel 0, 1 to channel 2, 3, respectively, in every other cycle. Then, channel 1 is switched to channel 2. The blocked numbers in Fig. 8 indicate the digits which will be switched. Different from FFT, during the j-th stage of FFT^{-1} (lower part of Fig. 8), (x_0, x_1) and (x_2, x_3) are loaded at the 0-th cycle, and the corresponding outputs are (x_0, x_{16}) and (x_1, x_{17}). Moreover, the channel switch operation is also different; we first switch channel 1 to 2. Then channel 0, 1 are switched to channel 2, 3, respectively, in every other cycle.

TABLE 5
Cycle requirement and estimation of the FMLM^3 architecture in terms of different operations (n denotes the number of butterfly structures).

FFT & FFT^{-1} Operations						
Operation (#)	P=32		P=64		P=128	
	$A \neq \sqrt{2}$	$A = \sqrt{2}$	$A \neq \sqrt{2}$	$A = \sqrt{2}$	$A \neq \sqrt{2}$	$A = \sqrt{2}$
Proposed architecture: $n = 1$						
CT (2)	104		203		459	
CT^{-1}(2)	107		206		462	
NCT (2)	104	106	203	205	459	461
NCT^{-1} (1)	109	111	208	210	464	466
Proposed architecture: $n = 2$						
CT (2)	79 (24%)		122 (39.9%)		235 (48.8%)	
CT^{-1}(2)	90		141		270	
NCT (2)	79	81	122	124	235	237
NCT^{-1} (1)	92	94	143	145	272	274
Estimation: $n = 4$						
CT (N.A.)	67 (35.6%)		94 (53.7%)		141 (69.3%)	
Estimation: $n = P/2$						
CT (N.A.)	60 (42.3%)		72 (64.5%)		84 (81.7%)	

Other Operations	
Operation (#)	Cycle requirement
Component-wise multi-add. (4)	$\Delta_{MA} + P + 3$
Time domain add. & sub. (7)	$P/2 + 6$ or $P/4 + 6$
Time domain div. by 2 (1)	$P/2 + 5$ or $P/4 + 5$

5.5 Clock Cycle and Estimation Analysis

FFT/FFT^{-1} is the most time consuming operation in the FMLM^3. Each FFT/FFT^{-1} computation is divided into v stages. The computation of each FFT/FFT^{-1} stage requires $\frac{P}{2} + \Delta_{FFT}$ cycles when using the single BFS architecture, where Δ_{FFT} is the pipeline depth of the *FFT/FFT*$^{-1}$ unit. In order to reduce the cycle requirement, an efficient way is to increase the number of BFSs so that more digits is performed at each cycle. As shown in Fig. 4, two BFSs are applied and four digits is processed in parallel, as a result, $\frac{P}{4}$ cycles is saved for each stage.

In this work, the designed pipeline depth of BFS and CS are $\Delta_{BFS} = 6$ and $\Delta_{CS} = 2$, respectively. Additionally, the read/write operation requires 3 cycles. Therefore, the cycle requirement of each stage is computed by:

$$\Delta_{non\_final\_stage} = \max\{\frac{P}{2n}, 11 + \frac{P}{4n}\} \qquad (28)$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2601334, IEEE Transactions on Computers

TABLE 6
Implementation results of the FMLM^3 on Virtex-6 (xc6vlx130t-1), gray colored rows refer to the architecture with 1 butterfly structure, the rest rows refer to the architecture with 2 butterfly structures

l	Algorithm (2 or 3)	M	P	u	LUTs	Slices	DSP	RAMB36 / RAM18	Cycles	Period (ns)	Latecy (μs)	Area-latency product (LUTs$\times\mu s$)
1,024	3	2^{128}+1	32	32	13,811	3,923	27	26 / 1	657	5.48	3.60	49,725
1,024	2	2^{96}+1	32	32	7,845	2,225	27	16 / 9	905	4.78	4.33	33,937
1,024	3*	2^{64}+1	64	16	6,047	1,757	9	16 / 1	1,052	3.80	4.00	24,173
1,024	3*	2^{64}+1	64	16	5,003	1,509	9	16 / 1	1,372	4.10	5.63	28,143
2,048	3	2^{224}+1	32	64	21,975	6,525	90	38 / 11	728	7.58	5.52	121,263
2,048	2	2^{160}+1	32	64	15,621	4,687	45	26 / 11	905	6.12	5.54	86,518
2,048	3*	2^{128}+1	64	32	13,975	3,928	27	28 / 1	1,061	5.34	5.67	79,179
2,048	3*	2^{64}+1	128	16	7,337	2,083	9	17 / 1	2,036	3.88	7.90	57,960
2,048	3*	2^{64}+1	128	16	5,964	1,720	9	22 / 0	2,945	4.07	11.99	71,485
3,072	2	2^{224}+1	32	96	23,460	6,909	90	38 / 11	918	7.72	7.09	166,260
3,072	2	2^{128}+1	64	48	13,684	3,931	27	28 / 1	1,462	5.24	7.66	104,831
3,072	2	2^{64}+1	128	24	7,351	2,103	9	22 / 1	2,770	3.84	10.64	78,191
3,072	2	2^{64}+1	128	24	5,838	1,696	9	35 / 1	4,131	3.98	16.44	95,985
4,096	2	2^{288}+1	32	128	34,995	10,257	135	50 / 11	917	9.47	8.68	303,575
4,096	2	2^{192}+1	64	64	19,005	5,525	81	39 / 2	1,474	6.88	10.14	192,723
4,096	3*	2^{128}+1	128	32	14,243	3,898	27	29 / 0	2,040	5.41	11.04	157,191
4,096	3*	2^{128}+1	128	32	11,898	3,529	27	34 / 0	2,945	5.71	16.82	200,076
7,680	2	2^{128}+1	128	60	14,683	4,248	27	34 / 0	2,774	5.38	14.92	219,131

* Corresponding parameter sets are suitable for both Algorithm 2 and 3, we use Algorithm 3 for a "free" FFT/FFT^{-1} reduction.

where n is the number of BFSs. The cycle requirement of the final stage Δ_{final_stage} is considered separately:
• For CT, the cycle requirement is $11 + \frac{P}{2n}$;
• For CT^{-1}, the digits from BFSs are forwarded to FSOs without passing through the signed bit correction operator and the CS block. Also note that the FSOs handle two digits per clock cycle which result in a 2 cycles delay for accumulation. To sum, the cycle requirement is $14 + \frac{P}{2}$;
• For NCT, when $A \neq \sqrt{2}$, the cycle requirement is $11 + \frac{P}{2n}$. When $A = \sqrt{2}$, two extra cycles are required and the cycle become $13 + \frac{P}{2n}$;
• For NCT^{-1}, when $A \neq \sqrt{2}$, the cycle requirement is $16 + \frac{P}{2}$. When $A = \sqrt{2}$, the cycle requirement is $18 + \frac{P}{2}$.
To summarize the above cases, the cycle requirement of FFT/FFT^{-1} is computed as:

$$FFT\_Cycles = (v-1)\Delta_{non\_final\_stage} + \Delta_{final\_stage}, \quad (29)$$

where Δ_{final_stage} are decided according to the transform type and value of A as discussed above.

Table 5 summarizes the cycle requirement of the proposed FMLM^3 architecture where Algorithm 2 with $P = 32$, $P = 64$ and $P = 128$ is applied. The cycle estimation for multiple BFS ($n = 4$ and $n = \frac{n}{2}$) are also included for reference. For simplicity, we only compares the cycle numbers of CT. Note that the modular R and Q' reductions is performed by addtion/subtraction in our design (cf. Section 5.3.1 and 5.3.2), the number of time domain addition/subtraction is counted as 7. Besides, the cycles of read-and-write operations are also included because our design works sequentially from the top-level view.

It can be observed from Table 5 that for the same P, the cycle requirement decreases with the increase of n, which leads to a larger cycle reduction. The percentages in the parentheses represent the cycle reduction from the multiple BFS ($n > 1$) to the corresponding single BFS. It worth to note that the cycle reduction is more obvious when the transform length P grows larger for the same n. Take $n = 2$ as an example, when $P = 32$, a 24% cycle reduction is achieved, while this percentage reaches 48.8% when $P = 128$.

6 Implementation Results and Comparisons

The proposed FMLM^3 architecture is implemented on a Virtex-6 (xc6vlx130t-1) FPGA. Synthesis and Place & Route are carried out by using Xilinx ISE 14.7 with default settings. Each DSP48E1 in Virtex-6 FPGA contains a signed 25×18-bit multiplier, and we use it to build the base multiplier unit of the *Multiply Adder*. The based multipliers are pipelined with the optimal stages for high frequency.

Parameter sets with 1024, 2048, 3072, 4096 and 7680-bit operand sizes are implemented. Table 6 presents the post place-and-route results for the selected parameters. The rows highlighted gray depict the implementation results with 1 BFS, while the rest of the rows depict with 2 BFSs.

In Table 6, the second column indicates the algorithm (Algorithm 2 or 3) applied to the corresponding parameter set. For the same transform length P and digit size u, the implementation of Algorithm 3 usually requires a larger M due to the different definitions of c in (15) and (20), which results in a larger area cost. For example, when $P = 32$ and $u = 32$, the first set has a larger M, so its implementation costs more look-up-table (LUTs) and has a longer critical path compared to the second one. On the contrary, the first implementation requires less clock cycles, this is because Algorithm 3 requires less NWTs than that of Algorithm 3.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2601334, IEEE Transactions on Computers

TABLE 7
Implementation comparison with the state-of-the-art architectures.

l	Platform	Design	DSP	DSP red. (%)	RAMB36[1]	RAM red. (%)	LUTs	Latency (μs)	Area-latency product (LUTs$\times\mu s$)	Area-latency efficiency improvement (%)
1,024	Virtex-II-4[2]	[14] (radix-2, $\omega = 32$)	—	—	—	—	5,310	10.09	53,577	—
1,024	Virtex-4	[31] ($\beta = 6$, $r = 64$)	—	—	—	—	30,467	2.16	65,689	—
1,024	Virtex-5	[13] (Arch2-V4, $k = 64$)	33	—	8	—	704	4.21	2,964	—
1,084	Virtex-6-1	[18] ($d = 128$, $\mu = 17$)	9	0	11	-50.0	4,818	7.57	36,472	50.9
1,024	Virtex-6-1	Our design (1 BFS)	9	0	16.5	0	5,003	5.63	28,143	16.4
1,024	Virtex-6-1	Our design (2 BFSs)	9	—	16.5	—	6,047	4.00	24,172	—
2,048	Virtex-II-4	[14] (radix-2, $\omega = 32$)	—	—	—	—	10,587	20.68	218,939	—
2,076	Virtex-6-1	[18] ($d = 64$, $\mu = 65$)	54	83.3	27.5	36.4	14,895	5.52	82,220	41.9
2,048	Virtex-6-1	Our design (1 BFS)	9	0	22	20.5	5,964	11.99	79,179	36.6
2,048	Virtex-6-1	Our design (2 BFSs)	9	—	17.5	—	7,337	7.90	57,960	—
3,072	Virtex-II-4	[14] (radix-2, $\omega = 32$)	—	—	—	—	15,197	30.94	470,195	—
3,196	Virtex-6-1	[18] ($d = 256$, $\mu = 25$)	9	0	11	-104.5	5,835	18.49	107,889	38.0
3,072	Virtex-6-1	Our design (1 BFS)	9	0	35.5	36.6	5,838	16.44	95,985	22.8
3,072	Virtex-6-1	Our design (2 BFSs)	9	—	22.5	—	7,351	10.64	78,191	—
4,096	Virtex-II-4	[14] (radix-2, $\omega = 32$)	—	—	—	—	19,621	41.85	821,138	—
4,124	Virtex-6-1	[18] ($d = 64$, $\mu = 129$)	135	80.0	49.5	41.4	27,839	7.78	216,587	37.8
4,096	Virtex-6-1	Our design (1 BFS)	27	0	34	14.7	11,898	16.82	200,076	27.3
4,096	Virtex-6-1	Our design (2 BFSs)	27	—	29	—	14,243	11.04	157,191	—
8,192	Stratix-V	[19] ($b = 16$)	72	—	463Kb	—	214,321	9.71	2,080,488	—
7,740	Virtex-6-1	[18] ($d = 128$, $\mu = 121$)	81	66.7	44	22.7	30,230	14.73	445,287	103.2
7,680	Virtex-6-1	Our design (2 BFSs)	27	—	34	—	14,683	14.92	219,131	—

1: RAM18 is counted as 0.5 RAMB36; 2: Virtex-II-4 denotes Virtex-II FPGA with speed grade 4.

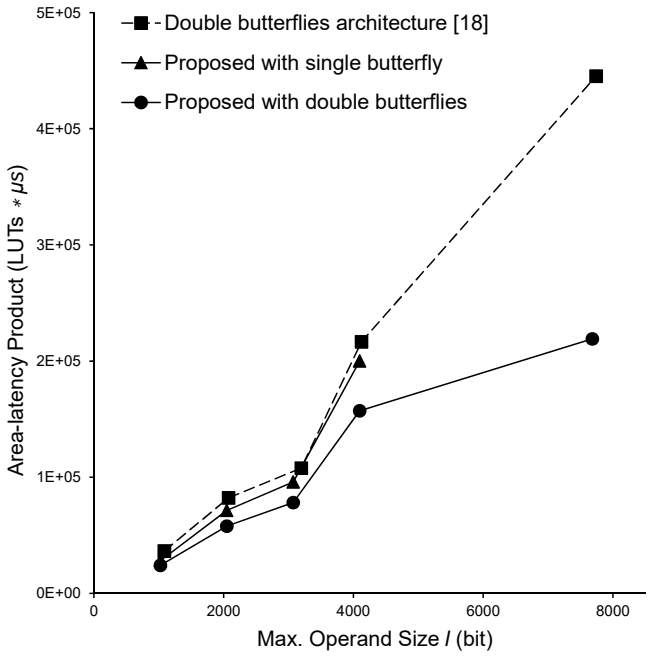

Fig. 9. Area-latency product comparison with [18].

The performance comparison between the FMLM[3] and the state-of-the-art architectures is provided in Table 7. We only select the most efficient parameter set for each operand size l for comparison. The design in [14] and [13] implement the digit-based method, [31] implements the RNS method, and [18] and [19] implement FFT method. Since different design may involve different DSP and RAM resources, their costs are also considered rather than only evaluating the area-latency product. In order to provide a fair comparison, the reduction and improvement ratios of [13], [14], [19] and [31] are not included, since they are implemented by other FPGA families.

In Table 7, the first column shows our design could fit the NIST recommended key size perfectly, while in [18], larger operand sizes are always required (i.e., when targeting 1024-bit key size, the l of our design is exactly 1024-bit compared to that of 1084-bit in [18]). A larger operand size may lead to waste of hardware resources. Our design has lower area-latency products compared to [18] for all the operand sizes, with an average of 54% area-latency efficiency improvement. The area-latency product growth tendency is also provided in Fig. 9 including the design of [18] and ours with 1 and 2 BFSs. It can be observed that our design with 2 BFSs has the least growth rate, which implies that the area-latency efficiency improvement becomes more obvious for larger operand sizes. For 1024 and 3072-bit cases, both [18] and our design employ the same number of DSP blocks, but our design employs more RAM blocks. This is because the FMLM[3] has more precomputations. Moreover, an extra RAM set is included to enable parallel computing of Step 8 and Step 9 in Algorithm 2 (cf. Section 5.4). For 2048, 4096, and 7680-bit cases, our design employs less DSP and RAM blocks with an average reduction ratios of 72% and 35%, respectively. The reduction mainly comes from the low hardware complexity cause by the small ring size. Besides, according to the comparison results, architecture with 2 BFSs also has a better area-latency efficiency, and less DSP

and RAM usage than that of 1 BFS.

Additionally, it can be observed from Table 7 that the design of [13] achieves a very small area-latency product, but employs 33 DSP blocks. The designs of [31] and [14] employs no DSP and RAM resources, but they have relatively large area-latency products. The design of [19] requires only 463Kb memory bits for case $l = 7680$, while the 34 RAM blocks used in our design is capable for more than 1Mb memory bits. This is mainly because the FMLM^3 includes more pre-computations than the algorithm of [19]. However, [19] costs more DSP blocks and its area-latency product is about 10 times greater than ours.

7 Conclusions

In this work, we proposed a modified version of the FFT-based Montgomery modular multiplication algorithm under McLaughlin's framework (FMLM^3). By applying cyclic and nega-cyclic convolutions to compute the modular multiplication steps, the zero-padding operation is avoided and the transform length is reduced by half compared to the regular FFT-based multiplication. Furthermore, we explored for some special cases, the number of transforms can be further reduced from 7 to 5 without extra computational efforts, so that the FMLM^3 can be further accelerated. A general method of efficient parameter set selection has been summarized for a given operand size.

Moreover, pipelined architectures with 1 and 2 butterfly structures are designed for high area-latency efficiency. We also analysed the connection between the number of butterfly structures and the cycle requirement. The estimation results indicate a feasible physical approach can be implemented which could trade area cost for faster speed by adding more butterfly structures. The Virtex-6 FPGA implementation results shows the proposed FMLM^3 with both 1 and 2 butterfly structures have better area-latency efficiency than the state-of-the-art FFT-based Montgomery modular multiplication. In addition, the processing speed of the proposed multiplier is also comparable, especially for large transform length (i.e. $P = 64$ or higher).

Acknowledgments

This work was supported by the Research Grant Council of the Hong Kong Special Administrative Region, China (Projects No. CityU 111913, CityU 123612) and Croucher Statup Allowance, 9500015.

References

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[2] R. L. Rivest, "A description of a single-chip implementation of the RSA cipher," *Lambda*, vol. 1, no. Fourth Quarter, pp. 14–18, 1980.

[3] E. Barker, W. Barker, W. Burr, W. Polk, M. Smid, P. D. Gallagher *et al.*, "NIST special publication 800-57 recommendation for key management–part 1: General," 2012.

[4] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[5] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," in *Soviet physics doklady*, vol. 7, 1963, p. 595.

[6] S. A. Cook and S. O. Aanderaa, "On the minimum computation time of functions," *Transactions of the American Mathematical Society*, pp. 291–314, 1969.

[7] A. Schönhage and V. Strassen, "Schnelle multiplikation großer zahlen," *Computing*, vol. 7, no. 3-4, pp. 281–292, 1971.

[8] M. Fürer, "Faster integer multiplication," *SIAM Journal on Computing*, vol. 39, no. 3, pp. 979–1005, 2009.

[9] D. Harvey, J. Van Der Hoeven, and G. Lecerf, "Even faster integer multiplication," *arXiv preprint arXiv:1407.3360*, 2014.

[10] S. Covanov and E. Thomé, "Fast arithmetic for faster integer multiplication," *arXiv preprint arXiv:1502.02800*, 2015.

[11] A. F. Tenca and Ç. K. Koç, "A scalable architecture for modular multiplication based on Montgomery's algorithm," *Computers, IEEE Transactions on*, vol. 52, no. 9, pp. 1215–1221, 2003.

[12] M. D. Shieh and W. C. Lin, "Word-based Montgomery modular multiplication algorithm for low-latency scalable architectures," *Computers, IEEE Transactions on*, vol. 59, no. 8, pp. 1145–1151, 2010.

[13] M. Morales-Sandoval and A. Diaz-Perez, "Scalable gf (p) montgomery multiplier based on a digit–digit computation approach," *IET Computers & Digital Techniques*, 2015.

[14] M. Huang, K. Gaj, and T. El-Ghazawi, "New hardware architectures for Montgomery modular multiplication algorithm," *Computers, IEEE Transactions on*, vol. 60, no. 7, pp. 923–936, 2011.

[15] G. C. Chow, K. Eguro, W. Luk, and P. Leong, "A Karatsuba-based Montgomery multiplier," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE, 2010, pp. 434–437.

[16] M. K. Jaiswal and R. C. C. Cheung, "Area-efficient architectures for large integer and quadruple precision floating point multipliers," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 2012, pp. 25–28.

[17] J. P. David, K. Kalach, and N. Tittley, "Hardware complexity of modular multiplication and exponentiation," *Computers, IEEE Transactions on*, vol. 56, no. 10, pp. 1308–1319, 2007.

[18] D. D. Chen, G. X. Yao, R. C. C. Cheung, D. Pao, and Ç. K. Koç, "Parameter space for the architecture of FFT-based Montgomery modular multiplication," *Computers, IEEE Transactions on*, vol. 65, no. 1, pp. 147–160, 2016.

[19] W. Wang and X. Huang, "A novel fast modular multiplier architecture for 8,192-bit RSA cryposystem," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. IEEE, 2013, pp. 1–5.

[20] D. S. Phatak and T. Goff, "Fast modular reduction for large wordlengths via one linear and one cyclic convolution," in *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*. IEEE, 2005, pp. 179–186.

[21] G. Saldamlı and Ç. K. Koç, "Spectral modular exponentiation," in *Computer Arithmetic, 2007. ARITH'07. 18th IEEE Symposium on*. IEEE, 2007, pp. 123–132.

[22] P. McLaughlin Jr, "New frameworks for Montgomerys modular multiplication method," *Mathematics of Computation*, vol. 73, no. 246, pp. 899–906, 2004.

[23] R. Crandall and B. Fagin, "Discrete weighted transforms and large-integer arithmetic," *Mathematics of Computation*, vol. 62, no. 205, pp. 305–324, 1994.

[24] H. J. Nussbaumer, *Fast Fourier transform and convolution algorithms*. Springer-Verlag Berlin Heidelberg, 1982.

[25] J. M. Pollard, "The fast Fourier transform in a finite field," *Mathematics of computation*, vol. 25, no. 114, pp. 365–374, 1971.

[26] G. X. Yao, J. Fan, R. C. C. Cheung, and I. Verbauwhede, "Novel rns parameter selection for fast modular multiplication," *Computers, IEEE Transactions on*, vol. 63, no. 8, pp. 2099–2105, 2014.

[27] N. Smart, S. Babbage, D. Catalano, C. Cid, B. d. Weger, O. Dunkelman, and M. Ward, "ECRYPT II yearly report on algorithms and keysizes (2011-2012)," *European Network of Excellence in Cryptology (ECRYPT II), Sept*, 2012.

[28] R. Creutzburg and M. Tasche, "Number-theoretic transforms of prescribed length," *Mathematics of computation*, vol. 47, no. 176, pp. 693–701, 1986.

[29] R. Zimmermann, "Efficient VLSI implementation of modulo $(2^n \pm 1)$ addition and multiplication," in *Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on*. IEEE, 1999, pp. 158–167.

[30] D. J. Bernstein, "Multidigit multiplication for mathematicians," *Advances in Applied Mathematics*, pp. 1–19, 2001.

[31] D. Schinianakis and T. Stouraitis, "Multifunction residue architectures for cryptography," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 61, no. 4, pp. 1156–1169, 2014.