FPGA-based Accelerators of Fully Pipelined Modular Multipliers for Homomorphic Encryption

Sunwoong Kim*, Keewoo Lee†, Wonhee Cho†, Jung Hee Cheon†‡, and Rob A. Rutenbar§
*Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
†Department of Mathematical Sciences, Seoul National University, Seoul 08826, South Korea
‡CryptoLab, Seoul 08826, South Korea
§Department of Computer Science and Department of Electrical and Computer Engineering,
University of Pittsburgh, Pittsburgh, PA 15260, USA
Email: sunwoong@illinois.edu, {activecondor, wony0404, jhcheon}@snu.ac.kr, rutenbar@pitt.edu

***Abstract*—Homomorphic encryption (HE) is an important cryptographic primitive which allows privacy preserving computations. Current HE schemes are all based on modular arithmetic. Modular multiplication (ModMult) is one of the most frequently used modular operations, but in practice it is often prohibitively slow due to a reduction operation with high computational complexity. To address this speed problem, we demonstrate a set of novel FPGA-based accelerators for fully pipelined ModMults in this paper. For a high-throughput integer multiplier (IntMult) in the ModMult designs, digital signal processing (DSP) slices on FPGAs are efficiently exploited with optimized IntMult designs. For the full RNS-HEAAN scheme, which is our target HE scheme, our proposed Barrett ModMult design is optimized using specific moduli and extended to the Shoup ModMult algorithm.**

Our proposed Barrett and Shoup ModMult designs implemented on a Xilinx Virtex UltraScale FPGA show a 2× shorter delay, 14× higher throughput at the same frequency, and 3× higher throughput/DSP than the previous non-fully pipelined Barrett ModMult design on average. In particular, our Barrett ModMult design with the specific moduli shows the highest throughput/DSP value although precomputation required in the Shoup ModMult design is not used. Compared with a reference software implementation, our ModMult designs show 679× faster average processing speeds when we deploy multiple ModMult cores that fully use DSP slices on our target FPGA.

***Index Terms*—Pipeline, FPGA, digital signal processing (DSP), modular multiplier, homomorphic encryption (HE).**

I. Introduction

Recently, deep neural networks have demonstrated outstanding solutions to various applications such as speech recognition and image classification. However, they require vast amount of open, i.e., non-private, unauthorized data for both training and inference. Therefore, practical approaches to so-called "privacy-preserving" machine learning, which allow personal data to be used with trust, are of critical and rising interest. Homomorphic encryption (HE) techniques, which allow computation on encrypted data, are in some sense an ideal solution for privacy-preserving machine learning [1]- [3]. One of the most promising HE schemes is the Homomorphic Encryption for Approximate Numbers (HEAAN), which is also called the CKKS scheme [4]. Its value comes from its effectiveness in real life problems. For example, the winner of 2017 Secure Genome Analysis Competition held by iDASH Privacy & Security Workshop (and every participant of 2018 workshop) used HEAAN for their solutions. Furthermore, the full residue number system (RNS)-based variant of HEAAN [5] is being supported by various tools such as Microsoft's HE library SEAL [6], IBM's HE library HElib [7], and an HE compiler CHET [8]. However, the RNS-HEAAN scheme still suffers from prohibitively slow processing speed, which renders it impractical for many critical HE-based applications. This impracticality stems from the fact that its intrinsic data types are polynomials with a high degree and large moduli, which are needed to guarantee the necessary level of security.

Modular multiplication (ModMult)[1] is one of the most frequently used operation in HE-based applications but time-consuming due to an expensive division operation. To alleviate its high complexity, various ModMult algorithms have been proposed to replace divisions by several integer multiplications (IntMults)[2]. The most popular reduction algorithms are Montgomery reduction [9] and Barrett reduction [10]. The conventional Montgomery ModMult is performed on a transformed domain and so efficient only when the transformation is relatively efficient than the requested operations, which is not our case. Therefore, we focus on the Barrett reduction as in the previous works [11]- [15].

A number of prior efforts have presented FPGA-based accelerators of ModMults for cryptosystems [12]- [21]. To increase the processing speed, they apply various approaches to the ModMult designs such as digital signal processing (DSP) slices on FPGAs that are efficient blocks for fast multiplication and accumulation, and a fully pipelined architecture for high throughput. However, to the best of our knowledge, a DSP-based fully pipelined Barrett ModMult design optimized using several specific moduli has never been demonstrated. Ozturk *et al.* proposed a DSP-based Barrett ModMult design, but it is not a fully pipelined design [13]. Cousins *et al.* presented a fully pipelined Barrett ModMult design but did not discuss how to use DSP slices efficiently [14].

The main contributions of this paper are as follows.

- We propose a DSP-based fully pipelined IntMult design. In this design, multiple DSP slices for each partial

[1]ModMult is also used to refer to a modular multiplier in this paper.
[2]IntMult is also used to refer to an integer multiplier in this paper.

978-1-7281-1957-1/19/$31.00

multiplication work in a parallel and pipelined manner, and each partial operand is put in a particular order to be of help to optimizations.
- We optimize the IntMult design by removing unnecessary hardware resources when only partial results are used, thereby reducing the numbers of look-up tables (LUTs), flip-flops (FFs), and DSP slices by 77%, 66%, and 17%, respectively. In addition, since partial operands are forwarded between IntMult designs, the delay and the number of FFs in the Barrett ModMult design are further reduced by 11% and 6%, respectively.
- We suggest an optimized Barrett ModMult design for specific moduli. Furthermore, we suggest a new working parameter for RNS-HEAAN with *lightweight* primes. The modulus and its scaled inverse with the *lightweight* primes both have low Hamming weight signed-binary expressions. Our list of the *lightweight* primes can also be used for RNS-FV scheme [22] or other schemes. The simplified IntMult design with this parameter reduces the number of DSPs and the delay by 63% and 42% respectively in our Barrett ModMult design.
- We apply our techniques to the Shoup ModMult algorithm [23] and show that our Barrett and Shoup ModMult designs are advantageous in terms of throughput/DSP for our specific *lightweight* moduli and for arbitrary moduli, respectively.
- We show that the proposed Barrett ModMult design using our *lightweight* moduli is 1149× faster than the software implementation of the ModMult in RNS-HEAAN [24] when a data bandwidth is assumed to be sufficient and all DSP slices on our target FPGA are used for ModMult.

The remainder of this paper is organized as follows. Section II provides background for this paper. Sections III and IV present the proposed Barrett ModMult design and its variants in an HE scheme, respectively. Section V evaluates the proposed ModMult designs, and Section VI concludes this paper.

II. Background

This section introduces the two critical ModMult algorithms, Barrett ModMult and Shoup ModMult. First, we define three IntMults used in the ModMult algorithms as follows.

- Full-IntMult: n-bits × n-bits → $2n$-bits
- Upper half (UH)-IntMult: n-bits × n-bits → upper n-bits
- Lower half (LH)-IntMult: n-bits × n-bits → lower n-bits

They are mentioned in the remainder of this paper.

A. Barrett Modular Multiplication

A basic principle of the Barrett reduction is to subtract the multiplication result between the quotient $\lfloor A/q \rfloor$ and the divisor (modulus) q from the input number A as shown in (1).

$$A \pmod q = A - \lfloor \frac{A}{q} \rfloor \times q \tag{1}$$

Algorithm 1 shows the ModMult algorithm using the Barrett reduction [25]. This algorithm includes three IntMults. The first IntMult (IntMult1) is a full-IntMult and multiplies two input operands, A and B (line 1). The other IntMults (IntMult2 and IntMult3) are additional multipliers for Barrett reduction. IntMult2 multiplies the upper bits of the output of IntMult1 (V) (line 2) by T that is scaled $1/q$ (line 3). Since only the upper bits of the output of IntMult2 (W) are multiplied by a modulus q in IntMult3 (line 4), IntMult2 is implemented using an UH-IntMult. Meanwhile, only the lower bits of the output of IntMult3 (X) are subtracted by those of IntMult1 (Y) (lines 6-10), and therefore IntMult3 is implemented using an LH-IntMult. Lastly, depending on the magnitude of the result Z, the update stage subtracts $2q$, q, or 0 from Z (lines 11-15). Fig. 1 shows a Barrett ModMult architecture with the three IntMults [13]. In this architecture, a shift register is used to delay the lower bits of the output of IntMult1 (Y), which accounts for a large portion of FFs in the ModMult design.

Algorithm 1 Barrett modular multiplication [25]

Input: A, B, q, $l = \lfloor \log_2 q \rfloor + 1$, $T = \lfloor 2^{2l}/q \rfloor$
Output: $A \times B \pmod q$

1: $U \leftarrow A \times B$
2: $V \leftarrow U \gg (l-1)$
3: $W \leftarrow (V \times T) \gg (l+1)$
4: $X \leftarrow W \times q \pmod{2^{l+1}}$
5: $Y \leftarrow U \pmod{2^{l+1}}$
6: **if** $X < Y$ **then**
7: $\quad Z \leftarrow 2^{l+1} + X - Y$
8: **else**
9: $\quad Z \leftarrow X - Y$
10: **end if**
11: **if** $Z \geq 2 \times q$ **then**
12: $\quad Z \leftarrow Z - 2 \times q$
13: **else if** $Z \geq q$ **then**
14: $\quad Z \leftarrow Z - q$
15: **end if**
16: **return** Z

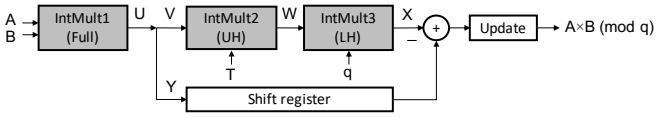


Fig. 1. Barrett ModMult architecture [13].

B. Shoup Modular Multiplication

Algorithm 2 shows the Shoup ModMult algorithm that is presented in the NTL library [23]. It is basically similar to the Barrett ModMult algorithm. However, compared with the conventional Barrett ModMult algorithm that approximates $1/q$ to $T/2^{2l}$ online, the Shoup ModMult algorithm precomputes the $B^{'}$ value that is a scaled approximation to B/q offline, based on the fact that the input number B and the modulus q are invariant. Using the $B^{'}$ value, $A \times B/q$ is approximated to W in Algorithm 2 [26]. In addition, the Shoup ModMult algorithm involves two LH-IntMults (lines 1 and 3) and a single UH-IntMult (line 2). Therefore, the Shoup

Algorithm 2 Shoup modular multiplication [23]

Input: A, B, q, $l = \lfloor \log_2 q \rfloor + 1$, $B' = \lfloor (B \ll (l+1))/q \rfloor$
Output: $A \times B \pmod q$

1: $X \leftarrow A \times B \pmod{2^{l+1}}$
2: $W \leftarrow (A \times B') \gg (l+1)$
3: $Y \leftarrow W \times q \pmod{2^{l+1}}$
4: $Z \leftarrow X - Y$
5: **if** $Z \geq q$ **then**
6: $\quad Z \leftarrow Z - q$
7: **end if**
8: **return** Z

ModMult algorithm has lower computational complexity than the conventional Barrett ModMult algorithm. One example of the use of the Shoup ModMult algorithm is a butterfly unit for a number theoretic transform (NTT) in the NTL library. Since the ModMult in the butterfly unit uses a constant input number, called the twiddle factor or the root of unity, and a constant modulus, the NTL library precomputes the B' values in advance and reuses them to increase the speed of NTT.

III. Fully Pipelined Barrett Modular Multiplier

In this section, our proposed fully pipelined Barrett ModMult hardware architecture is presented. To be specific, a DSP-based fully pipelined IntMult architecture that is employed in the ModMult design is proposed first. Next, optimized half-IntMult architectures and a forwarding scheme to reduce hardware resources and a delay are presented. In this paper, the input bit-width of ModMult designs is set to 62 because it is used in our target HE scheme, RNS-HEAAN [5]. However, the proposed design can be extended to various bit-widths.

A. DSP-based Fully Pipelined Integer Multiplier Architecture

A multiplier block embedded in a DSP slice on FPGAs has input bit-width limits. For example, input operands of the multiplier block in the DSP48E2 slice are 27-bit and 18-bit signed integers [27]. Therefore, a large operand should be divided into several partial operands to use DSP slices. The 62-bit Barrett ModMult needs a single 62-bit full-IntMult and two 63-bit half-IntMults. To use DSP48E2 slices for the 62-bit (or 63-bit) IntMult, the first operand is divided into a single 10-bit (or 11-bit) and two 26-bit partial operands. On the other hand, the second operand is divided into a single 11-bit (or 12-bit) and three 17-bit partial operands. Therefore, 12 DSP slices (three partial operands $\times$ four partial operands) are needed to perform the partial multiplications in parallel.

Fig. 2 shows the processing order of the proposed 62-bit IntMult using DSP48E2 slices. In this figure, A and B are 62-bit unsigned input operands, and A_i and B_j represent their partial operands ($i = 0, 1, 2$ and $j = 0, 1, 2, 3$). Each partial multiplication is performed by a single DSP slice, and 12 DSP slices for Stage1 through Stage12 work in a parallel and pipelined manner. To obtain the final multiplication result from the least significant bit (LSB), the divided operands are put into DSP slices in a particular order (A_0B_0, A_0B_1, A_1B_0, ..., A_1B_3, A_2B_2, A_2B_3). Lower bits of the partial multiplication results, which are represented by gray colors in Fig. 2, are used as the final multiplication results. To synchronize all partial multiplication results, shift registers are used. The 12 stages have different depths of the shift registers due to the pipelined architecture. For example, Stage1 has the deepest shift register while Stage12 does not exploit a shift register. On the other hand, higher bits of the partial multiplication results are forced to enter into the next stage. It means that the higher bits are accumulated from Stage1 to Stage12. For example, as represented by a dotted red rectangle in Fig. 2, higher 34-bits of the partial multiplication result in Stage8 (denoted by accumulated A_1B_2[42:9]) are added to the partial multiplication result between A_2 and B_1 in Stage9. Note that internal registers in a DSP slice, which is represented by rectangles with diagonal lines, are actively utilized to reduce the number of registers storing partial input operands for each DSP slice. Using this fully pipelined architecture, successive multiplication results are obtained every cycle after a delay.

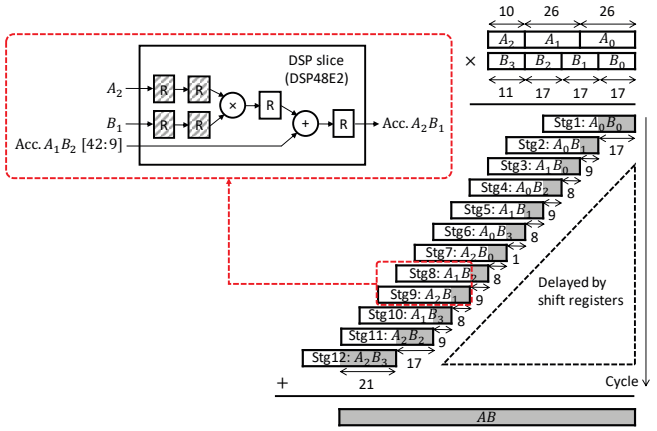


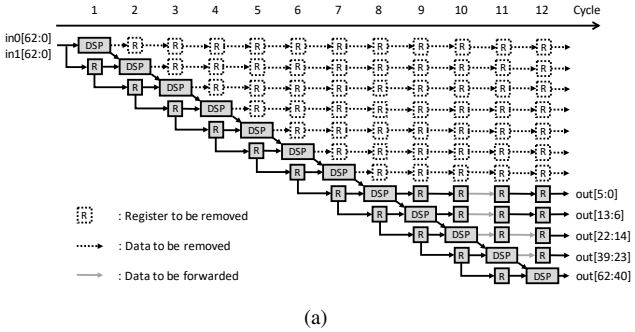
Fig. 2. Processing order of the IntMult architecture using DSP48E2 slices.

Even though the proposed (full) IntMult design is similar to the Xilinx IntMult LogiCORE IP [28] in that it is a fully pipelined design, it requires less DSP slices than the Xilinx IP (see Table II). In addition, it is able to provide a larger bit-width than 64 that is the maximum bit-width provided by the Xilinx IntMult IP. Lastly, the proposed processing order where multiplication outputs are generated from the LSB is efficiently used in optimizations for half-IntMults, which is presented in the next subsection.

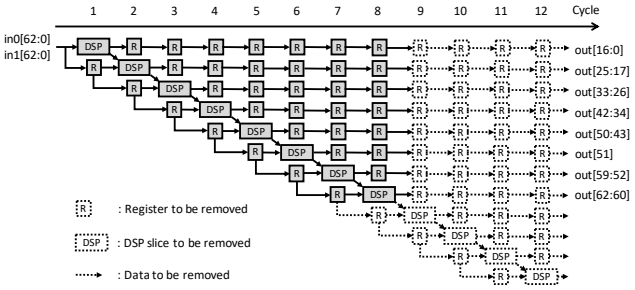
B. Optimized Half Integer Multipliers and Forwarding Scheme

Our Barrett ModMult design is based on the previous architecture shown in Fig. 1. However, it is a fully pipelined design thanks to our proposed IntMult design. In this subsection, the proposed IntMult design is optimized to reduce hardware resources and a delay of the Barrett ModMult design.

Fig. 3 shows optimized 63-bit half-IntMult architectures where horizontal and vertical axes represent the cycle and the pipeline stage, respectively. Basically, there are 12 DSP slices for the 12 stages, registers for partial input operands,



(a)



(b)

Fig. 3. Optimized half-IntMult architectures. (a) UH-IntMult (b) LH-IntMult.

and shift registers to delay partial multiplication results. As shown in Fig. 3(a), the UH-IntMult design outputs only upper 63-bits of the multiplication result, and therefore shift registers for Stage1 through Stage7, represented by dotted rectangles, are removed. It allows the UH-IntMult design to have a better property in the number of registers as well as that of DSP slices than the Xilinx IntMult IP (see Table II). On the other hand, lower 63-bits of the multiplication result are only outputted from the LH-IntMult design, so the number of pipeline stages in the LH-IntMult design decreases from 12 to 8, as shown in Fig. 3(b). As a result, four DSP slices and several shift registers that are represented by dotted rectangles are removed, and the delay is reduced by four cycles.

Suppose that the input bit-width of the proposed Barrett ModMult design is represented by b_I, and the larger (e.g. 26-bits) and smaller (e.g. 17-bits) bit-widths of a multiplier block in a DSP slice are represented by b_L and b_S, respectively. Then, the total number of DSP slices in the UH-IntMult design (and the full-IntMult design), denoted as n_{UH}, and that in the LH-IntMult design, denoted as n_{LH}, are calculated as follows.

$$n_{\text{UH}} = \lceil \frac{b_I + 1}{b_L} \rceil \times \lceil \frac{b_I + 1}{b_S} \rceil \tag{2}$$

$$n_{\text{LH}} = \sum_{k=0}^{\lfloor (b_I+1)/b_L \rfloor} \lceil \frac{b_I - b_L \times k + 1}{b_S} \rceil \tag{3}$$

Using (2) and (3), the total number of DSP slices in the proposed Barrett ModMult design can be calculated. For example, when the b_I values are 45 and 55, 17 ($= 6{+}6{+}5$) and 31 ($= 12 + 12 + 7$) DSP48E2 slices are needed, respectively.

The optimization reduces hardware resources and/or delay inside half-IntMult designs. Now, an additional optimization

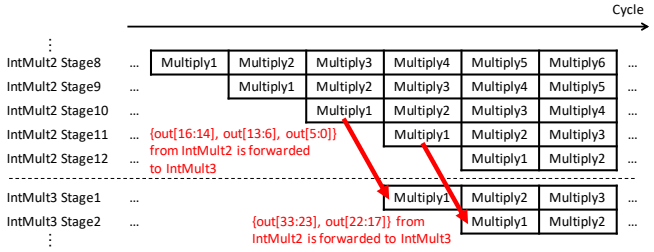


Fig. 4. Forwarding the first and second 17-bit data from IntMult2 (UH-IntMult) to IntMult3 (LH-IntMult).

that is achieved when the proposed IntMult designs are successively connected is described. Suppose that the 63-bit results of UH-IntMult are put into the 17-bit input operand slot B of a multiplier block in a DSP slice for LH-IntMult. As represented by gray arrows in Fig. 3(a), the least significant 17-bits of the result of UH-IntMult (out[16:0]) are generated at Cycle 10, which means that they can be used as the partial input operand B_0 of LH-IntMult at the next cycle. Therefore, in our Barrett ModMult design, the intermediate 17-bit data are forwarded from Stage10 of IntMult2 to Stage1 of IntMult3 as shown in Fig. 4. Similarly, the next 17-bits generated from IntMult2 (out[33:17]), which are the partial input operand B_1 of IntMult3, are forwarded to Stage 2 of IntMult3. This forwarding scheme is also applicable between IntMult1 and IntMult2 of the Barrett ModMult design. As well as the reduction in delay, the data forwarding between IntMult2 and IntMult3 reduces the depth of the shift register shown in Fig. 1, which affects the FF usage of the Barrett ModMult design. As the bit-width of input operands increases, the effectiveness of the proposed forwarding scheme (e.g. the numbers of cycles and FFs reduced by the forwarding scheme) increases because lower bits of the output, which are partial input operands to the next IntMult, are available relatively early.

Note that synthesis and place-and-route stages of Xilinx Vivado Design Suite remove unnecessary registers and/or DSP slices of the Xilinx IntMult IP. However, the amounts are less than ours, and data forwarding does not occur in the synthesis and place-and-route stages (see the third-fourth rows and sixth-seventh rows of Table II).

IV. Variants in Homomorphic Encryption

In this section, we optimize the proposed ModMult design using characteristics of HE schemes. Specifically, the proposed ModMult design is modified using specific moduli with low Hamming weights and extended to the Shoup ModMult algorithm that is fast and area-efficient in a specific situation. We note that the optimization techniques presented in this section focus on the RNS-HEAAN scheme [5], but it can be applied to other conventional HE schemes as well.

A. Simplified Integer Multiplier With Specific Moduli

In some previous works, a ModMult design adopts a customized reduction scheme using a specific modulus such as the Solinas modulus [29], $q = 2^{64} - 2^{32} + 1$, and replaces IntMults

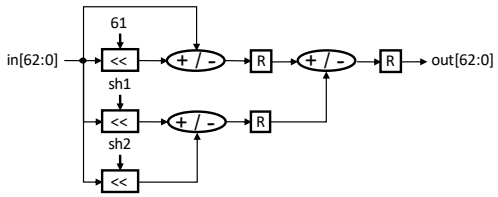


Fig. 5. Simplified half-IntMult architecture for specific moduli with the maximum Hamming weight of 4.

for reduction with several additions and/or subtractions [12], [17], [18], [20]. However, there are limitations in choosing such specific moduli. The main problem is that the Solinas prime itself is too rare, so that it is not feasible to find a list of Solinas primes with extra properties we need. Furthermore, the ModMult design cannot be shared by another moduli, and therefore it is inappropriate in HE schemes that use several different moduli.

For an area-efficient ModMult design that is shared by several moduli, the existing q and T values, which are shown in Algorithm 1, are replaced with combinations of only power-of-two numbers. For example, q is represented by $2^{61} \pm 2^{\mathrm{sh}_1} \pm 2^{\mathrm{sh}_2} \pm ... \pm 2^{\mathrm{sh}_{h-2}} \pm 1$ when the Hamming weight is h. The details on how to find the specific q and T values in our target HE scheme, RNS-HEAAN, are described in the following subsection. The specific q and T values enable IntMult3 (LH-IntMult) and IntMult2 (UH-IntMult) in our Barrett ModMult design to be simply implemented using only shifters and adders/subtractors, respectively.

Fig. 5 shows the simplified half-IntMult architecture. In this architecture, the specific q and T values have the Hamming weight less than or equal to 4, and the number of pipeline stages is 2. Since the maximum Hamming weight is 4, a 63-bit input number A generates four terms in Stage1: $A \ll 61$, $A \ll \mathrm{sh}_1$, $A \ll \mathrm{sh}_2$, and A. The $A \ll \mathrm{sh}_i$ value is set to 0 if the term 2^{sh_i} is 0. After that, the first and second terms are added to or subtracted by the fourth and third terms, respectively. In Stage2, the final result is generated by the rightmost adder/subtractor using the two intermediate results. Depending on the type of half-IntMult, the simplified half-IntMult design outputs upper or lower 63-bits. This design that does not use a DSP slice is helpful for an area-efficient design on FPGAs because the number of DSP slices is usually much more limited than the numbers of LUTs and FFs.

B. Moduli Selection in RNS-HEAAN

One of the most promising HE schemes is the HEAAN algorithm suggested by Cheon *et al.* in ASIACRYPT'17 [4]. Unlike other HE schemes, HEAAN naturally supports approximate arithmetic with fixed point numbers. This remarkable feature provides much more efficient HE-based applications for real life problems such as machine learning algorithms [30] and cyber-physical systems [31]. More recently, a full RNS-based variant of HEAAN was suggested [5]. This scheme decomposes a large modulus of the original scheme into small and similar-sized primes to be more friendly to conventional 64-bit processors. Thanks to this, the scheme is efficient in general computing systems, but the efficiency can be further improved by hardware acceleration, having large potential parallelism of ModMults.

In the RNS-HEAAN scheme, mainly three types of moduli are used: Base modulus, Rescale moduli, and Modup moduli. The base modulus is the last remaining modulus after all calculation processes, and its size is generally 10-bit larger than the scale factor that is used to preserve the precision of a message in the RNS-HEAAN scheme. The rescale moduli serve to preserve the precision of a message after multiplication, and its size is similar to the scale factor. The modup moduli are used to reduce the size of errors in the multiplication process, and its size is independent from the scale factor. The size of primes for the modup moduli is set considering factors related to the efficiency of the RNS-HEAAN scheme. For instance, the RNS-HEAAN scheme uses 62-bit primes for 64-bit processors. These three types of moduli should satisfy the following first condition (called NTT condition) for correctness and/or efficiency of the RNS-HEAAN scheme. For details, readers are referred to [5].

1) When the degree of polynomials is $N-1$, primes should be congruent to 1 mod $2N$, which must be guaranteed for a speed up in NTT-based polynomial multiplication.

The current reference software implementation of RNS-HEAAN with N of 2^{15} uses one 55-bit prime for the base modulus, eleven 45-bit primes for the rescale moduli, and twelve 62-bit primes for the modup moduli [24]. When an integer A is multiplied by the moduli in this software implementation, this multiplication cannot be completely replaced with shifters and adders. To optimize ModMults in the RNS-HEAAN scheme by applying our method presented in Section IV-A, we propose a new working parameter for RNS-HEAAN, composed of primes satisfying the following second condition (called lightweight condition) as well as the first condition.

2) The prime q and its scaled inverse T both have low Hamming weight signed-binary expressions. This is to completely replace IntMults with shifters and adders in the ModMult design for speedup.

To find the list of primes satisfying the two conditions mentioned above, we made an elementary but useful observation. That is, if an integer q is represented by the signed binary form with the minimum Hamming weight (i.e. by a non-adjacent form [25]) as $2^l \pm 2^{\mathrm{sh}_1} \pm 2^{\mathrm{sh}_2} \pm ... \pm 2^{\mathrm{sh}_{h-2}} \pm 1$, where $l > 2 \cdot \mathrm{sh}_1 + 1$, then the scaled inverse T corresponding to the q value can be represented as $2^l \mp 2^{\mathrm{sh}_1} \mp 2^{\mathrm{sh}_2} \mp ... \mp 2^{\mathrm{sh}_{h-2}} \mp 1$ with the same Hamming weight and reversed signs. The fact comes from the following inequality.

$$(2^l \pm 2^{\mathrm{sh}_1} \pm ... \pm 2^{\mathrm{sh}_{h-2}} \pm 1) \cdot (2^l \mp 2^{\mathrm{sh}_1} \mp ... \mp 2^{\mathrm{sh}_{h-2}} \mp 1)$$
$$= 2^{2l} - (\pm 2^{\mathrm{sh}_1} \pm ... \pm 2^{\mathrm{sh}_{h-2}} \pm 1)^2 > 2^{2\ell} - q$$

Using this property, we can find the list of (l+1)-bit primes satisfying the two conditions much more efficiently. Instead of performing exhaustive search all over the interval ($2^l - 2^{l-1}$, $2^l + 2^{l-1}$) and checking if the candidate and its scaled inverse

TABLE I
TWELVE 62-BIT q AND T (HAMMING WEIGHT ≤ 4) FOR MODUP MODULI

Index	q	T
1	$2^{61} - 2^{26} + 1$	$2^{61} + 2^{26} - 1$
2	$2^{61} - 2^{24} - 2^{20} + 1$	$2^{61} + 2^{24} + 2^{20} - 1$
3	$2^{61} - 2^{24} + 1$	$2^{61} + 2^{24} - 1$
4	$2^{61} - 2^{22} + 2^{19} + 1$	$2^{61} + 2^{22} - 2^{19} - 1$
5	$2^{61} - 2^{21} + 1$	$2^{61} + 2^{21} - 1$
6	$2^{61} - 2^{21} + 2^{16} + 1$	$2^{61} + 2^{21} - 2^{16} - 1$
7	$2^{61} + 2^{22} + 2^{20} + 1$	$2^{61} - 2^{22} - 2^{20} - 1$
8	$2^{61} + 2^{23} - 2^{18} + 1$	$2^{61} - 2^{23} + 2^{18} - 1$
9	$2^{61} + 2^{23} + 2^{21} + 1$	$2^{61} - 2^{23} - 2^{21} - 1$
10	$2^{61} + 2^{24} - 2^{19} + 1$	$2^{61} - 2^{24} + 2^{19} - 1$
11	$2^{61} + 2^{25} + 2^{23} + 1$	$2^{61} - 2^{25} - 2^{23} - 1$
12	$2^{61} + 2^{26} + 2^{16} + 1$	$2^{61} - 2^{26} - 2^{16} - 1$

both have small Hamming weights, we search over the interval $(2^l - 2^{l/2},\ 2^l + 2^{l/2})$, where the condition $l > 2 \cdot \mathrm{sh}_1 + 1$ holds, and check if only the candidate itself has a small Hamming weight, which is quadratically better. Moreover, using the method, the size of primes for the rescale moduli is automatically similar to the scale factor. For example, our list of (l+1)-bit primes can be seen as approximations of 2^l with at most $2^{l/2}$ errors.

As a result, we found one 55-bit prime with the Hamming weight of 3 for the base modulus, eleven 45-bit primes with the Hamming weight of 5 for the rescale moduli, and twelve 62-bit primes with the Hamming weight of 4 for the modup moduli, satisfying the two conditions. Table I shows the twelve 62-bit primes for our *lightweight* moduli. This new parameter is helpful for fast and area-efficient hardware implementation of RNS-HEAAN without harming the functionality of the original scheme.

C. Application to Shoup Modular Multiplication

ModMults in the RNS-HEAAN scheme are categorized into two groups. The first group receives two arbitrary input numbers. On the other hand, one of input numbers of the second group is an invariant and known integer. The twiddle factor or the root of unity in a butterfly unit of NTT and inverse NTT is an example of the invariant input number.

The Shoup ModMult algorithm has advantages on FPGAs. First, as shown in Algorithm 2, the Shoup ModMult algorithm moves a part of calculations to offline when one of input numbers and a modulus are constants. On FPGAs, the precomputed values can be stored in internal memory such as BRAMs and reused. Secondly, the Shoup ModMult algorithm replaces the full-IntMult in the Barrett ModMult algorithm with the LH-IntMult, thereby reducing the number of DSP slices when arbitrary moduli are used. It enables more ModMult designs to be deployed on an FPGA.

To further validate effectiveness of our proposed techniques, we apply them to the Shoup ModMult algorithm. Fig. 6 shows the Shoup ModMult architecture where IntMult1 is implemented using the proposed UH-IntMult design while IntMult2 and IntMult3 are implemented using the proposed LH-IntMult design. When the *lightweight* moduli presented in Section IV-B are used, the LH-IntMult for IntMult3 is replaced with the simplified LH-IntMult shown in Fig. 5. Compared with the Barrett ModMult design, the delay of the Shoup ModMult design is shorter because the IntMult1 and IntMult2 of the Shoup ModMult work in parallel. Thanks to the reduced delay, the depth of the shift register is reduced as well. In Section V-B, we compare the Shoup ModMult design with the Barrett ModMult design and analyze which design is better in cases when arbitrary and specific moduli are used, respectively.

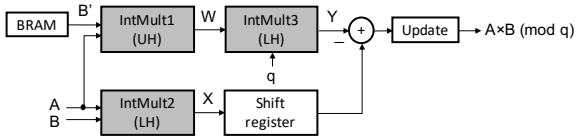


Fig. 6. Shoup ModMult hardware architecture.

V. EVALUATION

In this section, we evaluate our proposed hardware designs on an FPGA. All the previous and proposed new designs are implemented in Verilog HDL, and the Xilinx Vivado Design Suite (2016.4) is used for synthesis and place-and-route stages. The target device is the Xilinx Virtex UltraScale FPGA (xcvu190-flgc2104-2-e) that has 1M LUTs, 2M FFs, and 1800 DSP slices. For comparison with a software implementation, we implement on an Intel Xeon processor E5-2643 v4 running at 3.4 GHz with a 20MB cache.

In general homomorphic applications, there are many successive modulo operations, which are the same, due to the large polynomial degree N, e.g. 2^{15}. In addition, frequencies for input/output (I/O) and memory interfaces are typically faster than that for computing logic blocks. Therefore, it is assumed that a delay generated by I/O and/or memory interfaces is negligible compared with the total latency, and successive I/O and/or memory access times are hidden by a pipelined schedule.

A. Comparison of Fully Pipelined Integer Multipliers

We first evaluate fully pipelined IntMult designs. Table II compares the proposed IntMult designs in our 62-bit Barrett ModMult designs with the Xilinx IntMult LogiCORE IP [28]. The Xilinx IP is generated with optimum settings. All results in this table are obtained after the place-and-route stage. As shown in the second and fifth rows, the number of FFs used in our full-IntMult design is twice as many as that in the Xilinx IntMult design due to the shift registers placed after DSP slices. Instead, the proposed design requires four less DSP slices and a 5 cycle shorter delay. Therefore, it is useful for deploying many IntMults on FPGAs where the number of DSP slices is much fewer than the numbers of LUTs and FFs. Compared with our full-IntMult design, the optimized UH-IntMult design reduces the numbers of LUTs and FFs by 88% and 60%, respectively. On the other hand, the optimized LH-IntMult design reduces the number of DSP slices and the delay by 4 and 4 cycles respectively, as well as reducing the

TABLE II
COMPARISON OF XILINX LOGICORE IP AND OUR INTMULTS IN THE 62-BIT BARRETT MODMULT DESIGNS (AFTER PLACE-AND-ROUTE)

	Type	LUTs	FFs	DSPs	Delay (cycles)
Xilinx IntMult [28]	62b full	216	478	16	18
	63b UH	186	418	16	18
	63b LH	130	290	10	18
Proposed IntMult	62b full	161	962	12	13
	63b UH	20	381	12	$13(-2)^{\dagger}$
	63b LH	53	281	8	$9(-2)^{\dagger}$
	63b UH rev	806	279	0	2
	63b LH rev	379	194	0	2

†Additional reduction in delay by our proposed forwarding scheme

TABLE III
COMPARISON OF 62-BIT BARRETT AND SHOUP MODMULT HARDWARE DESIGNS FOR MULTIPLE MODULI ON AN FPGA

	Ozturk *et al.* [13]	**Proposed (single-core)**			
		D1	**D2**	**D3**	**D4**
Algorithm	Barrett	Barrett	Barrett	Shoup	Shoup
$f_{\max}$ (MHz)	367	375	425	400	450
LUT	1117	708	1988	364	689
FF	1090	1971	1810	1186	1102
DSP	3	32	12	28	20
Delay (Cycle)	47	33	19	22	17
ModMult/Cycle	0.07	≈ 1	≈ 1	≈ 1	≈ 1
ModMult/(μs·DSP)	8.16	11.72	35.42	14.29	22.50

numbers of LUTs and FFs. When the half-IntMult designs in our ModMult designs receive partial operands early by our proposed forwarding scheme, their delays are additionally reduced by 2 cycles. Compared with the Xilinx half-IntMult LogiCORE IPs that are generated by using options in the Xilinx Vivado Suite, our half-IntMult designs use less hardware resources and show shorter delay. As shown in the last two rows, our simplified half-IntMult designs using our *lightweight* moduli that are presented in Section IV-B utilize many LUTs for adders/subtractors. However, they do not use DSP slices, and reduce the delay to 2 cycles, which has a great effect on hardware implementation results for our ModMult designs.

B. Comparison of Modular Multiplier Hardware Designs

To evaluate our techniques for ModMult, we present implementation results for the four ModMult designs below:

- D1: Barrett ModMult for arbitrary 62-bit moduli
- D2: Barrett ModMult for specific 62-bit moduli
- D3: Shoup ModMult for arbitrary 62-bit moduli
- D4: Shoup ModMult for specific 62-bit moduli

For the specific moduli, the parameter set in Table I is used.

Table III shows FPGA implementation results of 62-bit ModMult designs for multiple moduli. The second through seventh rows present the ModMult algorithm, maximum frequency, the numbers of LUTs, FFs, and DSP slices, and delay. The eighth row shows the ModMult/Cycle value. Since our ModMult designs are fully pipelined designs, their ModMult/Cycle values are 1. The ninth row presents the ModMult/(μs·DSP) value. It is the main metric in this paper because it refers to how efficiently DSP slices are used for throughput. Note that BRAMs that are required for $B^{'}$ of the Shoup ModMult algorithm, shown in Algorithm 2, are not included in this table.

As a previous work, the Barrett ModMult design for arbitrary moduli presented by Ozturk *et al.* [13] is used. This ModMult design includes three DSP-based IntMult designs that work in a parallel and pipelined manner. However, each IntMult design iteratively exploits a single DSP slice, which means that the ModMult design is not a fully pipelined design. For fair evaluation, the previous ModMult design that originally receives 32-bit input numbers is extended to a 62-bit design. Since the 62-bit IntMult design has more partial multiplications than the 32-bit design, it increases the delay and reduces the throughput.

Although not presented in Table III, the Barrett ModMult design proposed by Cousins *et al.* is a fully pipelined design, where the ModMult/Cycle value is almost 1, for arbitrary moduli [14]. However, it inherently requires more hardware resources than our designs, even though the detailed results are not presented in [14]. For example, it divides a 64-bit ModMult into four 16-bit $\times$ 64-bit sub-ModMults and runs them in a parallel and pipeline manner. Each of the sub-ModMults has 47 internal pipeline stages, and therefore the total delay of the ModMult design is 188 cycles, which is about 8 times longer than those of our designs, and many shift registers are needed to hold input numbers. In addition, three 64-bit full-IntMults are used for each sub-ModMult, so twelve 64-bit full-IntMults are deployed in total for the ModMult design.

As shown in the second and third columns of Table III, the proposed D1 design shows slightly higher frequency, 37% lower LUT count, and 30% shorter delay than the previous design. Even though it exploits more DSP slices for fully pipeline processing, it increases the ModMult/(μs·DSP) value by 44%. Compared with the D1 design, the D2 design with our *lightweight* moduli increases the maximum frequency by 13%. As shown in the fourth row, the number of LUTs increases because three adders/subtractors shown in Fig. 5 are used instead of DSP slices in each half-IntMult design. However, the number of DSP slices and the delay are reduced by 63% and 42%, respectively. The improved frequency and reduced DSP count results in a 3$\times$ higher ModMult/(μs·DSP) value. The proposed Shoup ModMult designs, D3 and D4, significantly reduce the numbers of LUTs and FFs of the D1 and D2 designs, respectively, because the full-IntMult is replaced with the LH-IntMult and the depth of the shift register decreases. In addition, the delays of the D1 and D2 designs are reduced by 11 cycles and 2 cycles, respectively, because two half IntMults, IntMult1 and IntMult2 in Fig. 6, work in parallel. Compared with the D1 design, the D3 design increases the maximum frequency by 7% and reduces the number of DSP slices by 13%. It implies that the Shoup ModMult design is more advantageous than the Barrett ModMult design for arbitrary moduli. On the other hand, the D4 design uses 8 more DSP slices than the D2 design because the UH-IntMult is not replaced with the simplified half-IntMult. Therefore,

TABLE IV
EXECUTION TIME OF MODMULT IMPLEMENTATIONS

Software implementation of Barrett ModMult [24]			6.49ms
The proposed FPGA implementations (single)	D1	-	0.96ms
	D2	-	0.85ms
	D3	-	0.90ms
	D4	-	0.80ms
The proposed FPGA implementations (multi)	D1	core no. = 56	17.17μs
	D2	core no. = 150	5.65μs
	D3	core no. = 64	14.08μs
	D4	core no. = 90	8.90μs

if our *lightweight* moduli are available, the Barrett ModMult design is better than the Shoup ModMult design in terms of ModMult/(μs·DSP). Furthermore, the Barrett ModMult design does not need precomputation and BRAMs for $B^{'}$ required in the Shoup ModMult design.

C. Comparison with Software Implementation of RNS-HEAAN

Table IV compares our FPGA implementations with a reference software implementation of the Barrett ModMult in the open source code of RNS-HEAAN [24]. Input operations for evaluation are set to successive $2^{15} \times 11$ ModMults. Since our designs are fully pipelined designs, they generate one result every cycle after a delay. Therefore, "the delay $+ 2^{15} \times 11$" cycles are required in our designs to complete the input operations. When a single ModMult core is deployed, as shown in the second through fifth rows, the D1, D2, D3, and D4 designs show 6.76×, 7.64×, 7.21×, and 8.11× faster processing speeds than the software implementation, respectively. The sixth through ninth rows show execution times when multiple ModMult cores are deployed. The number of cores is determined based on the total number of DSP slices on our target FPGA, which is 1800, and the number of DSP slices used in each design (e.g. D1 with 56 (=$\lfloor 1800/32 \rfloor$) cores). It is assumed that the I/O bandwidth is sufficient to put input numbers into the multiple cores and all logic blocks on the FPGA are used for the ModMult designs, which is ideal. Since multiple cores work in parallel, the speedup is proportional to the number of cores. All designs achieve the speedup of more than 378×. Specifically, the D3 design with 64 cores for arbitrary moduli and the D2 design with 150 cores for specific moduli accelerate the processing speed of the software implementation by 461× and 1149×, respectively.

VI. CONCLUSION

This paper proposes fully pipelined ModMult designs for an HE scheme on an FPGA. Specifically, it focuses on how to efficiently use DSP slices for high throughput. Our proposed designs show better throughput results than the previous design on an FPGA and achieve a significant speedup over a reference software implementation. In addition, we demonstrate that the Barrett ModMult design using our *lightweight* moduli is faster than the Shoup ModMult design. Since ModMult is one of the most frequently used operators in HE schemes, our proposed ModMult designs can make a significant contribution to practical HE-based applications. In future work, we plan to apply our ModMult designs to NTT and further apply to the homomorphic multiplication in the RNS-HEAAN scheme.

REFERENCES

[1] N. Dowlin *et al.*, "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy," in *Proc. ICML*, 2016.
[2] C. Juvekar *et al.*, "GAZELLE: A Low Latency Framework for Secure Neural Network Inference," in *Proc. USENIX Security*, 2018.
[3] F. Bourse *et al.*, "Fast Homomorphic Evaluation of Deep Discretized Neural Networks," in *Proc. CRYPTO*, 2018.
[4] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers," in *Proc. ASIACRYPT*, 2017.
[5] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A Full RNS Variant of Approximate Homomorphic Encryption," in *Proc. SAC*, 2018.
[6] H. Chen, K. Laine, and R. Player, "Simple Encrypted Arithmetic Library - SEAL v2.1," in *Proc. FC*, 2017.
[7] S. Halevi and V. Shoup, "Algorithms in HElib," in *Proc. CRYPTO*, 2014.
[8] R. Dathathri *et al.*, "CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing," in *Proc. PLDI*, 2019.
[9] P. L. Montgomery, "Modular Multiplication Without Trial Division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, 1985.
[10] P. Barrett, "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor," in *Proc. CRYPTO*, 1986.
[11] Y. Doroz *et al.*, "Accelerating Fully Homomorphic Encryption in Hardware," *IEEE Trans. Comput.*, vol 64. no. 6, pp.1509-1521, Jun. 2015.
[12] X. Cao *et al.*, "Optimised Multiplication Architectures for Accelerating Fully Homomorphic Encryption," *IEEE Trans. Comput.*, vol 65. no. 9, pp.2794-2806, Sep. 2016.
[13] E. Ozturk *et al.*, "A Custom Accelerator for Homomorphic Encryption Applications," *IEEE Trans. Comput.*, vol. 66, no. 1, Jan. 2017.
[14] D. B. Cousins *et al.*, "Accelerating Secure Computing with a Dedicated FPGA-based Homomorphic Encryption Co-Processor," *IEEE Trans. Emerging Topics Comput.*, vol. 5, no. 2, pp. 193-206, April-June 1, 2017.
[15] S. S. Roy *et al.*, "HEPCloud: An FPGA-Based Multicore Processor for FV Somewhat Homomorphic Function Evaluation," *IEEE Trans. Comput.*, vol. 67, no. 11, Nov. 2018.
[16] D. N. Amanor *et al.*, "Efficient Hardware Architectures for Modular Multiplication on FPGAs," in *Proc. FPL*, 2005.
[17] C. Du *et al.*, "High-Speed Polynomial Multiplier Architecture for Ring-LWE based Public Key Cryptosystems," in *Proc. GLSVLSI*, 2016.
[18] W. Wang and X. Huang, "FPGA Implementation of a Large-Number Multiplier for Fully Homomorphic Encryption," in *Proc. ISCAS*, 2013.
[19] T. Poppelmann *et al.*, "Accelerating Homomorphic Evaluation on Reconfigurable Hardware," in *Proc. CHES*, 2015.
[20] D. D. Chen *et al.*, "High-Speed Polynomial Multiplication Architecture for Ring-LWE and SHE Cryptosystems," *IEEE Trans. Circuits Syst. I*, vol. 62, no. 1, pp. 157-166, Jan. 2015.
[21] E. Ozcan and S. S. Erdem, "A Fast Digit based Montgomery Multiplier Designed for FPGAs with DSP Resources," *Elsevier Microprocessors and Microsystems*, vol. 62, pp. 12-19, Oct. 2018.
[22] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes," in *Proc. SAC*, 2016.
[23] "NTL: a library for doing number theory (Version 11.3.2)," [Online]. Available: http://www.shoup.net/ntl/
[24] "Source Code of A Full RNS Variant of HEAAN," [Online]. Available: https://github.com/HanKyoohyung/FullRNS-HEAAN/
[25] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. New York: Springer-Verlag, 2004.
[26] D. Harvey, "Faster Arithmetic for Number-Theoretic Transforms," *Journal of Symbolic Computation*, vol 60, pp. 113-119, Jan. 2014.
[27] Xilinx, "UltraScale Architecture DSP Slice v1.7 - UG579," Jun. 2018.
[28] Xilinx, "Multiplier v12.0 - PG108," Nov. 2015.
[29] J. A. Solinas, "Generalized Mersenne numbers," Faculty of Mathematics, Univ. Waterloo, Waterloo, Canada, Tech. Rep., 1999.
[30] A. Kim, Y. Song, M. Kim, K. Lee, and J. H. Cheon, "Logistic Regression Model Training based on the Approximate Homomorphic Encryption," *BMC Medical Genomics*, vol.11, Oct. 2018.
[31] J. H. Cheon *et al.*, "Toward a Secure Drone System: Flying With Real-Time Homomorphic Authenticated Encryption," *IEEE Access*, vol. 6, pp. 24325-24339, Mar. 2018.