New Integer-FFT Multiplication Architectures and Implementations for Accelerating Fully Homomorphic Encryption

Xiaolin Cao, Ciara Moore

CSIT, ECIT, Queen's University Belfast, Belfast, Northern Ireland, UK

xcao03@qub.ac.uk, cmoore50@qub.ac.uk

Abstract. This paper proposes a new hardware architecture of Integer-FFT multiplier for super-size integer multiplications. Firstly, a basic hardware architecture, with the feature of low hardware cost, of the Integer-FFT multiplication algorithm using the serial FFT architecture, is proposed. Next, a modified hardware architecture with a shorter multiplication latency than the basic architecture is presented. Thirdly, both architectures are implemented, verified and compared on the Xilinx Virtex-7 FPGA platform using 256, 512, 1024, 2048 and 8192 point Integer-FFT algorithm respectively with multiplication operands ranging from 2^{12} bits to 2^{25} bits in size. Experimental results show that the hardware cost of the proposed architecture is no more than 1/10 of the prior FPGA solution, and is perfectly within the implementable range of the Xilinx Virtex-7 FPGA platform, and outperforms the software implementations of the same bit-length operand multiplication on the Core-2 Q6600 and Core-i7 870 platforms. Finally, the proposed implementations are employed to evaluate the super-size multiplication in an encryption primitive of fully homomorphic encryption (FHE) over the integers. The analysis shows that the speed improvement factor is up to 26.2 compared to the corresponding integer-based FHE software implementation on the Core-2 Duo E8400 platform.

Keywords: Fully Homomorphic Encryption, FPGA, Hardware, Integer-FFT Multiplication.

1 Introduction

Super-size or million-digit integer multiplication plays an important role in cryptography. For example, it is extensively used in the schemes of fully homomorphic encryption (FHE) over the lattices [1, 2] and FHE over the integers [3 − 6]. A secure FHE scheme can be used to arbitrarily perform computations on a ciphertext, but without compromising the content of the corresponding plaintext. Therefore, a practical FHE scheme is thought to be the key module in numerous cloud-based security and privacy related applications, such as the privacy-preserving search, computing outsourcing and identity-preserving banking.

However, almost all existing reported FHE schemes [1 − 10] face severe efficiency and cost challenges, namely, impractical key sizes and a very large computational complexity. For example, Gentry and Halevi (GH) reported the first FHE scheme using the lattice theory with a public key size up to 2.3 Gigabytes, and a ciphertext homomorphic evaluation time of up to 30 minutes [11]. If this scheme is used to perform a homomorphic evaluation of the block cipher Advanced Encryption Standard (AES), it requires 36 hours to evaluate a single AES encryption operation [12]. Although the somewhat homomorphic encryption (SHE) requires a relatively shorter key size, it is still much heavier than a practical cipher scheme. For example, the SHE implementation presented by Lauter *et al*. employs a public key of 29 Kilobytes, and required an encryption time of 0.024 seconds [13].

Of the proposed FHE schemes to date, FHE over the integers has the advantage of comparatively simpler theory, as well as the employment of a much shorter public key, making its implementation somewhat more practical than that of other competing schemes. van Dijk *et al*. proposed the first scheme of FHE over the integers [3]. Then Coron *et al*. [4] in 2011 improved this scheme with a reduced public key size up to 802 MB, and reducing the encryption time to about 3 minutes. Next, the public key size is further reduced by Coron *et al*. [5] in 2012 to no more than 10.1 MB, but its software implementation takes a longer encryption time of 7 minutes. Moore *et al*. [14] analysed the possibility of super-size integer multiplication implementation of encryption primitives in integer-based FHE schemes by embedded DSP blocks on a FPGA platform using the Comba algorithm. In 2013, the batch processing technique is proposed by Cheon *et al*. [6] to improve the parallel computing ability of the scheme of FHE over the integers presented by van Dijk *et al*. [3].

In almost all these schemes, super-size multiplication is required due to the extremely large bit-length public and private keys. A typical multiplication algorithm for very large bit-length operands is Integer-FFT [15 − 18], as it has the smallest computation complexity of $O(nlog(n))$. For example, the widely used open-source GMP library uses Schönhage–Strassen Integer-FFT algorithm [15] for multiplication when the bit-length of operands is greater than 2^{15} bits [19]. Recently, a lot of efforts of using Integer-FFT to accelerate super-size multiplication have been reported in both hardware and software. Wang *et al*. reported the first software implementation on a NVIDIA C2050 GPU [20]. It uses the integer-FFT multiplication algorithm [15] to implement the super-size multiplication in Gentry's FHE scheme [2], and gained almost 7 times speed improvement over the work in [11]. Cousins *et al*. planed to obtain a scalable hardware Integer-FFT implementation on a FPGA platform using the Matlab® HDL Coder tool, however they do not report any implementation or simulation results yet [21, 22]. Poppelmann and Guneysu reported a hardware implementation of super-size polynomial multiplications using Integer-FFT algorithm which can be used in lattice-based FHE schemes [23]. The most recent progress is presented by Cao *et al*. [24]. With the Integer-FFT algorithm, they gave the first hardware implementation of the encryption primitives in integer-based FHE schemes of Coron *et al*. [4, 5]. It gains a significant speed accelerating factor 44.72 and 54.42 compared to the corresponding software implementations in [4, 5] respectively, however its hardware cost heavily exceeds the hardware budget of Xilinx Virtex-7 FPGA platforms.

The objective of this paper is to improve the prior work and to propose hardware architectures and implementations within the hardware budget of Xilinx FPGA plat-

form. Specifically, our contributions are as following. (i) A basic super-size hardware multiplier architecture using the Integer-FFT multiplication algorithm is proposed with the feature of lower hardware cost. (ii) A modified architecture is proposed to reduce the latency of the proposed basic architecture. (iii) The two proposed architectures are implemented and verified in Xilinx Virtex-7 FPGA using 256, 512, 1024, 2048 and 8192 point Integer-FFT algorithm with the operands up to 2^{25} bits. The experimental result shows our implementations run faster than their software counterparts on the Core-2 Q6600 and Core-i7 870 platforms, and only require no more than 1/10 FPGA hardware resource of the previous implementation. (iv) The multiplication operation in Coron *et al.*'s FHE scheme [5] is evaluated by the proposed implementations. The accelerating factor is up to 26.2 compare to the referenced software implementation.

The rest of the paper is organised as follows. In Section 2, the Integer-FFT multiplication algorithm is introduced. In Section 3 the proposed basic hardware architecture of the super-size multiplier is described. Next, Section 4 details the proposed modified architecture with a reduced latency. The implementation and performance comparison results are given in Section 5. Finally, Section 6 concludes the paper.

2 Review of The Integer-FFT Multiplication Algorithm

The Integer-FFT multiplication algorithm conquers very large bit-length multiplication by first dividing it into small bit-length multiplication, then accumulating and concatenating the small bit-length multiplication results to form the super-size product result. As the small bit-length multiplication requires more complexity than the addition, it is thought to be the bottleneck of improving the performance of the Integer-FFT algorithm. The Schönhage–Strassen Integer-FFT algorithm [15] relieves this problem through carefully selecting the FFT point count and the FFT modulus to make sure the twiddle factor used in the FFT is equal to 2. However, additional computational cost is incurred by the weighting and de-weighting operation in Schönhage–Strassen algorithm, and the very large FFT modulus also makes it unattractive for hardware implementations. In this paper, the basic Integer-FFT algorithm [16 − 18] is adopted. The reason is that the problem of accelerating the small bit-length multiplication speed can be solved with the help of the embedded multipliers in a Xilinx Virtex-7 FPGA device. These embedded multipliers can be automatically generated by Xilinx Core generator. The largest bit-length of automatically generated multiplier is up to 64, and the performance of an automatically generated pipelined 64-bit×64-bit multiplier is up to 300MHz [25].

The typical parameters of the Integer-FFT multiplication algorithm are as:

- p, a m-bit number, used as the modulus in the Integer-FFT modular reduction.
- k, the FFT point number.
- ω, the twiddle factor of the FFT.
- b, the base unit bit-length when transforming the input super-size operand into a b-bit digit sequence.

The necessary conditions of making sure that the Integer-FFT algorithm works correctly are that: (i) the FFT point number k divides $q - 1$ for every prime factor q of

p; (ii) the twiddle factor ω is a primitive k-th root of the finite field Z/pZ, which means that $\omega^k = 1\ (mod\ p)$ and $\omega^{k/q} - 1 \neq 0\ (mod\ p)$ for any prime divisor q of p [14], (iii) all operations used in the FFT should be modular with respect to the modulus, p; and (iv) the bit-length of multiplication product, z, is no more than kb.

We take an example, $z = x \times y$, to explain the Integer-FFT algorithm in this paper.

- Step-1: In this step, each super-size operand is transformed into a sequence of smaller (computationally efficient) bit-length numbers. Specifically, x is processed as a b-bit digit sequence, $\{x_t\}$ with $0 \leq t < k$, where x_0 to $x_{k/2-1}$ are filled by the real data bits of x from the least significant bit (LSB) to the most significant bit (MSB), and $x_{k/2}$ to x_{k-1} are filled with 0. Using the same method, y is processed to obtain $\{y_t\}$. Their relationship is expressed in Equation (1):

$$x = \sum_{t=0}^{k} x_t\ (mod\ p),\ y = \sum_{t=0}^{k} y_t\ (mod\ p) \tag{1}$$

- Step-2: Using the sequence $\{x_t\}$ as inputs to perform a k-point FFT to obtain a k-point sequence, $\{X_T\}$ with $0 \leq T < k$. The same operations are performed for $\{y_t\}$ to get the sequence $\{Y_T\}$. Equation (2) is used to describe this relationship.

$$X_T = \sum_{t=0}^{k-1} x_t\, \omega^{Tt}\ (mod\ p),\ Y_T = \sum_{t=0}^{k-1} y_t\, \omega^{Tt}\ (mod\ p) \tag{2}$$

- Step-3: Using the sequences $\{X_T\}$ and $\{Y_T\}$ as inputs to perform a point-wise multiplication, as in Equation (3), to get a k-point sequence $\{Z_T\}$ with $0 \leq T < k$:

$$Z_T = X_T \times Y_T\ (mod\ p) \tag{3}$$

- Step-4: Using the sequence $\{Z_T\}$ to perform a k-point IFFT, as in Equation (4), to get a k-point sequence $\{z_t\}$:

$$z_t = k^{-1} \sum_{T=0}^{k-1} Z_T\, \omega^{-tT}\ (mod\ p) \tag{4}$$

- Step-5: Perform a kb-bit addition to obtain the final product, $z = x \times y$, as described in Equation (5).

$$z = \sum_{t=0}^{k-1} \big(z_t \ll (t \times b)\big) \tag{5}$$

It must be noted that Step-1 in our described algorithm is a little different from the original Integer-FFT algorithm [15], in which if the bit-length sum of the two input operands, x and y, is no more than kb, then it just needs to fill the sequences $\{x_t\}$ and $\{y_t\}$ with all the real data bits of x and y from LSB to MSB. Therefore, it is necessary to fill 0 in sequences of $\{x_t\}$ and $\{y_t\}$, but not always necessary to fill 0 in the half of $\{x_t\}$ and $\{y_t\}$ as the described above. The reference papers could have different filling method [15 − 18], and the specific filling method is determined by the architecture and scheduling mechanism of the corresponding implementations.

3 The Proposed Basic Super-Size Multiplier Architecture

First of all, we take the same assumption as that in our prior proposal [24] as: there is sufficient off-chip memory available for the designed FPGA accelerator to store its intermediate variables and final results. This is a reasonable assumption as the accelerator could be viewed as a powerful coprocessor device, sharing memory with the main workstation (be it a server or PC) over a high speed PCI bus.

3.1 The Architecture Overview

An overview diagram of the proposed basic architecture of the super-size hardware multiplier is illustrated in Fig. 1. It consists of shared RAMs, an Integer-FFT module and a finite state machine (FSM) controller. The shared RAMs are assumed to be off-chip, and used to store the input operands, and intermediate and final results. The Integer-FFT module is the core module in our design. It is responsible of generating the multiplication result, and its architecture is illustrated in Fig. 2(i). It mainly consists of two FFT modules, one point-wise multiplication module, one IFFT module and one addition-recovery module. Basically, this architecture replicates the data flow explained in the algorithm described in Section 2.

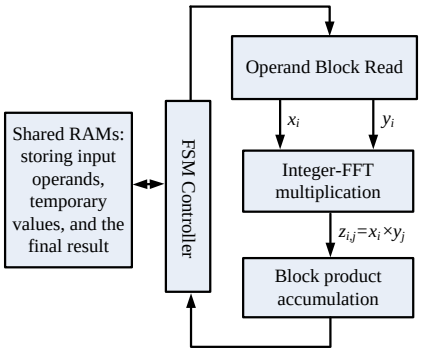


Fig. 1. The proposed super-size integer multiplication hardware architecture overview.

The FSM controller is responsible for distributing the controlling signals to schedule the Integer-FFT module, and it also implements an iterative school-book multiplication accumulation logic [15], which is illustrated in Fig. 2(ii), to accumulate the block products generated by the Integer-FFT module. For example in Fig. 2(ii), as the bit-length of multiplication operand, x and y, is too large, the multiplication cannot be completed by just a single Integer-FFT multiplication. x is divided into five data bit blocks from LSB to MSB, x_0 to x_4. y is divided into two data blocks from LSB to MSB, y_0 and y_1. Thus, the block multiplication iteration count is 10. In each iteration, the Integer-FFT module is called to compute a block product, $x_i \times y_j$ with $0 \leq i < 5$ and $0 \leq j < 2$. And the iteration can be divided into two level, namely, inner-iteration

and outer-iteration. The inner-iteration is used to iterate the data blocks of x, and the outer-iteration is to iterate the data blocks of y. Therefore, the proposed architecture can be viewed as a combination of school-book multiplication [15] and Integer-FFT multiplication described in Section 2.

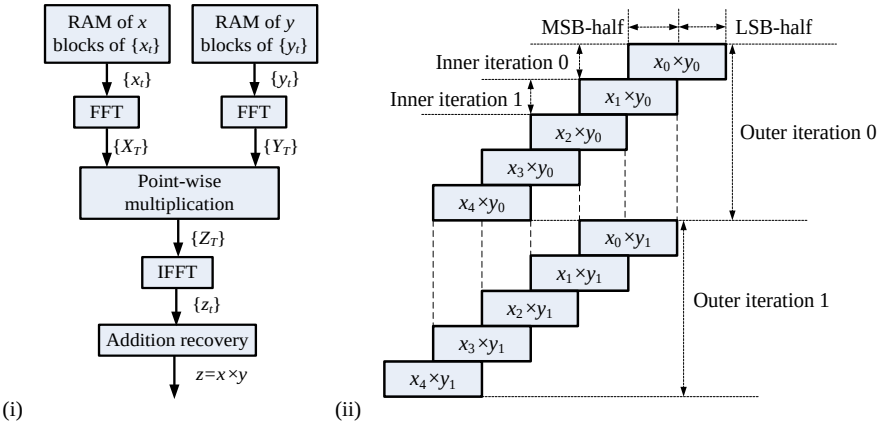


Fig. 2. (i) The proposed Integer-FFT module architecture used in the proposed basic architecture. (ii) The proposed block-accumulation logic used in the proposed basic architecture.

3.2 The FFT/IFFT Module and Their Butterfly Module

The most significant factor causing the very expensive hardware cost of the prior implementation [24] is due to the radix-2 fully parallel architecture for the FFT and IFFT. In that case, it requires $\log_2 k$ processing stages for a k-point FFT, and each processing stage is composed of $k/2$ parallel butterfly modules. Therefore, in this paper, we propose to use the serial FFT/IFFT architecture, which still requires $\log_2 k$ processing stages for a k-point FFT, but only one butterfly module in each processing stage. Therefore, the total butterfly module count can be reduced from $k/2 \times \log_2 k$ to $\log_2 k$.

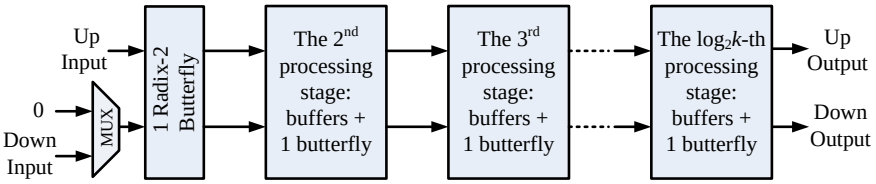


Fig. 3. The proposed serial FFT/IFFT architecture

The Fig. 3 illustrates the proposed serial architecture of the FFT/IFFT. The Down-Input is always equal to 0 in the FFT case. This can be explained by the FFT theory (the valid index distance between Up-Input and Down-Input is equal to $k/2$) [26] and filling 0 method in Step-1 in Section 2. Therefore, upon any valid input, the FFT

modules in Fig. 2(ii) do not need to wait for a valid Down-Input that requires $k/2$ operand reading clock cycles in order to compute a valid output.

As we have two FFT modules, and both of them read b-bit data each clock from the off-chip memory in parallel, the total bit-width of two FFT RAM data bus bit-width is equal to $2b$, and the total address bus bit-width of the two operand read ports is $\log_2 \frac{n_x}{b} + \log_2 \frac{n_y}{b}$, where it is assumed that the input operand is x and y, and their bit-length is n_x and n_y.

The processing stage architectures in both FFT and IFFT are illustrated in Fig. 4 and Fig. 5 respectively. Each processing stage consists of several buffers and one butterfly module. The buffer count of both up and down branches in each processing stage is the same. The difference is that the buffer count of the FFT processing stages decreases from $k/2$ to 1, but the buffer count of the IFFT processing stages increases from 1 to $k/2$. In some literature, this architecture is called radix-2 multi-path delay commutator (R2MDC) [26]. In most applications, only one FFT module is employed for data processing, and the decimator-in-frequency R2MDC is popularly used. In this paper, the decimator-in-time R2MDC is implemented both for FFT and IFFT.

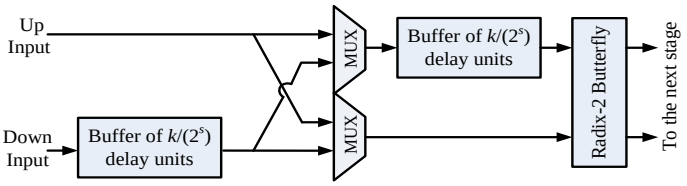


Fig. 4. The proposed s-th processing stage used in FFT

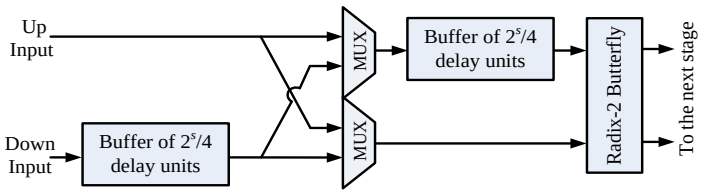


Fig. 5. The proposed s-th processing stage used in IFFT

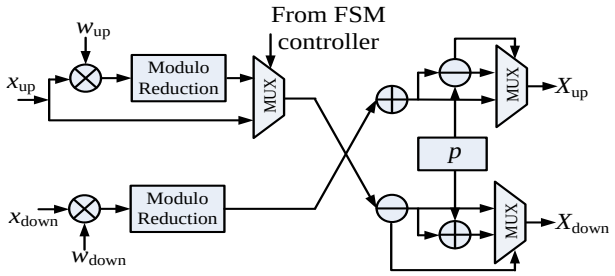


Fig. 6. The proposed radix-2 butterfly in each processing stage used in both of FFT and IFFT

The butterfly module proposed in this paper, illustrated in Fig. 6, is the same as our prior design [24]. The advantage of this architecture is that this design takes account into the requirement in both of FFT and IFFT, and makes both of FFT and IFFT share the same latency through pre-computing k^{-1} $(mod\ p)$ and incorporating it into the IFFT twiddle factors. Interested readers can check the details in [24].

In order to help the readers understand the data flow method in the proposed FFT/IFFT module, a 16-point (it means $k = 16$) FFT and IFFT example using the proposed architecture is illustrated in Fig. 7 and Fig. 8. The numbers plotted in both figures represent the position indices, whose ranges from 0 to 15 in each processing stage, of an input digit sequence to each butterfly module. From up to bottom in Fig. 7 and Fig. 8, a single 16-point sequence is processed clock by clock. Each clock, the position indices in the buffers is updated so that we can know how the data flows into and out of the FFT/IFFT module. It can be seen that both of the 16-point FFT and IFFT need four processing stage. The first processing stage only contains one butterfly module but no buffer. The FFT buffer count in these stages in both up and down branches is equal to 4, 2, and 1, and the IFFT buffer count in the same branch is the opposite order, 1, 2, and 4.

In the FFT module of Fig. 7, the first processing stage only needs its Up-Branch input, x_{up}, whose index range is from 0 to 7, and does not need any real data input as its Down-Branch input, x_{down}, whose index range is from 8 to 15, as it is always equal to 0. This also explains why the first processing stage does not require any buffer. The 2nd processing stage (it means $s = 2$ in Fig. 4) needs 8 buffers, 4 buffers in the Up-Branch and 4 buffers in the Down-Branch, to propagate the correct data pair into the butterfly module. For example, the data in position 0 needs to pair the data in position 2 according to the FFT theory [26] in this stage, so the minimum buffer count in both up and down branches is equal to 4. The reasoning behind the 3rd and 4th processing stages is the same. It can be found that the output data pairs of the FFT module has the similar properties to the input data pairs, that is, the Down-Branch index is equal to the sum of Up-Branch index adding 8.

In the IFFT module of Fig. 8, as the input data pairs of the IFFT module (i.e., the output data pairs of the FFT module) already have the correct indices, the 1st processing stage does not need any buffer. However, since the input data sequence of the IFFT module is not the natural order (i.e., 0, 1, 2, …) as the input of the FFT module, the 2nd processing stage requires 1 buffer in both the up and down branches to propagate correct data pairs to the butterfly module. The same reason results in that the subsequent 3rd and 4th processing stage need 2 and 4 buffer in each branch respectively. It must be noted that the output pair indices of the IFFT module has the properties that (i) the Down-Branch index is equal to the Up-Branch index plus 8, and (ii) the Up-Branch index increments from 0 to 7 by adding 1 each time. This property is utilised in the addition-recovery module design.

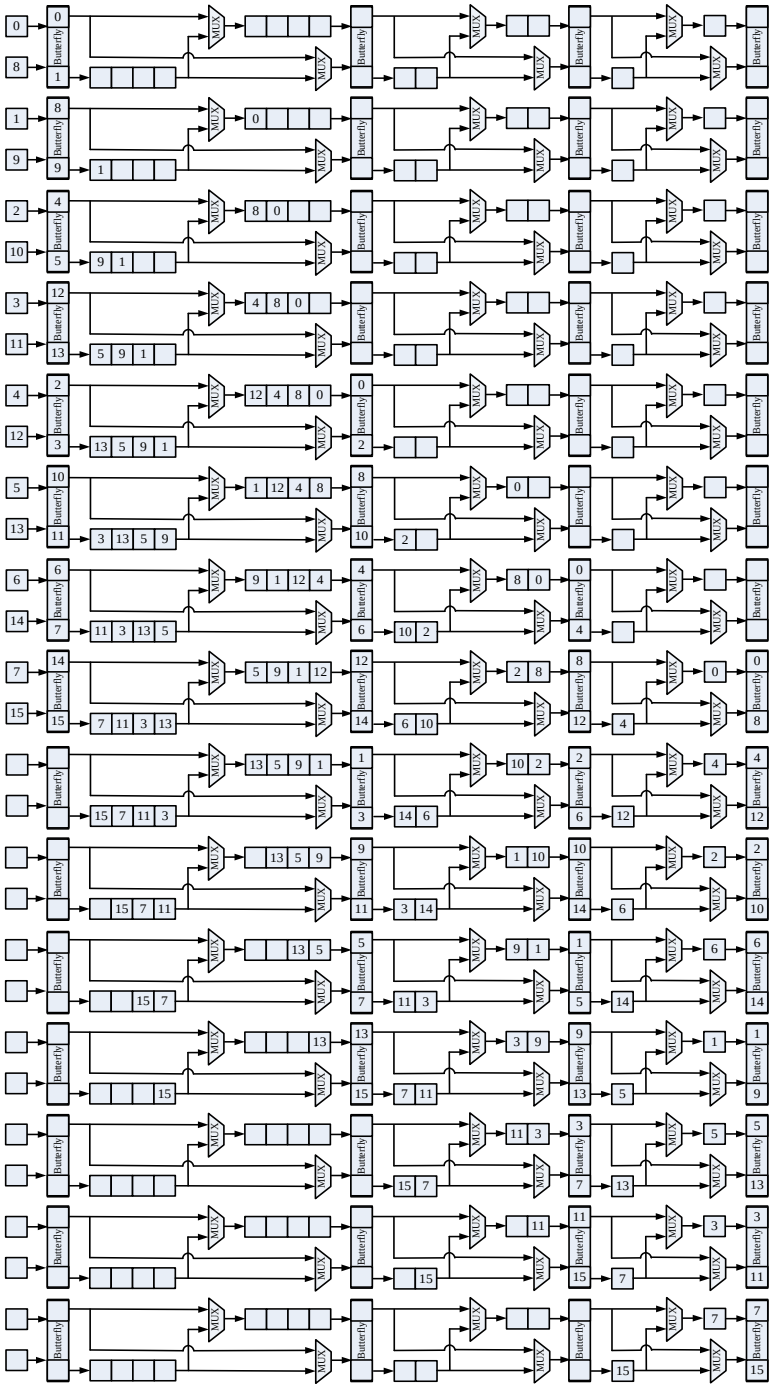


Fig. 7. The data-flow example when the proposed FFT is implemented by a 16-point FFT

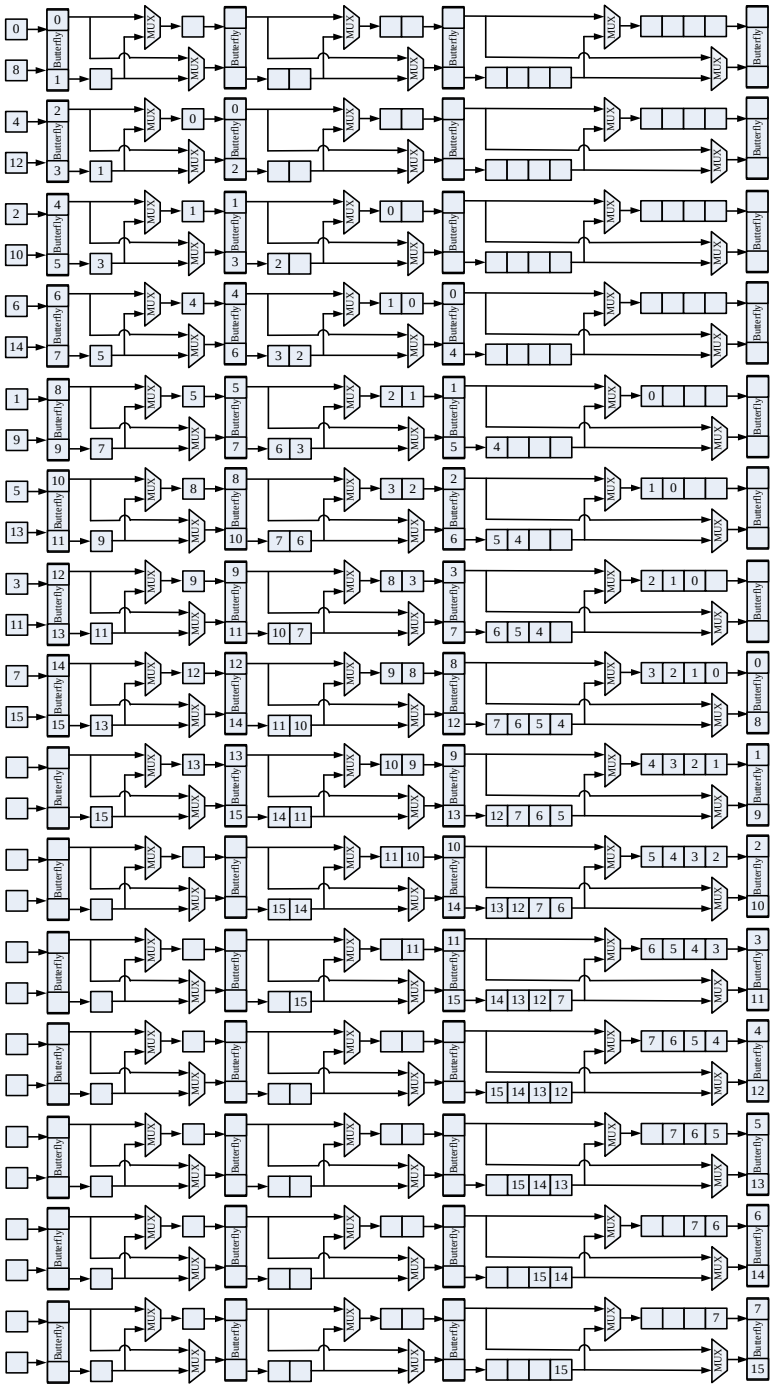


Fig. 8. The data-flow example when the proposed IFFT is implemented by a 16-point IFFT

3.3 The Point-Wise Multiplication Module

The Fig. 9 illustrates the proposed point-wise multiplication module. It consists of two parallel modular multiplication modules. The reason is that there are two FFT modules, and each of them has up and down branches outputs, X_{up}, Y_{up}, X_{down} and Y_{down}.

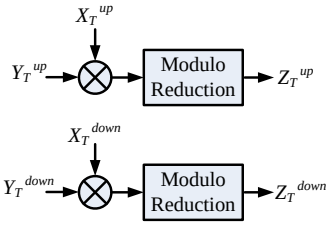


Fig. 9. The proposed point-wise multiplication module

Our prior implementation [24] utilises the fully parallel FFT architecture, so if a k-point FFT is used, it need k replicas of modular multiplication modules. Compared to our prior design, this new module also saves a lot of hardware resource, as the count of the employed modular multiplication modules is reduced from k to 2.

3.4 The Modular Reduction Module

This subsection introduces the modular reduction module used after the multiplication operation in FFT/IFFT butterfly and point-wise multiplication modules. The addition/subtract modular reduction only needs subtraction and additions with the modulus, and is already illustrated in the right-half part in Fig. 6.

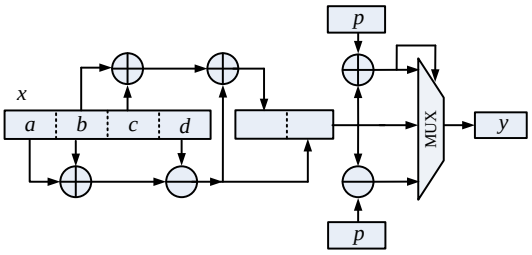


Fig. 10. The proposed modular reduction used in the proposed FFT/IFFT butterfly and point-wise multiplication modules

Our prior work pointed out that the selection of a reasonable modulus, p, heavily influences the modular multiplication performance in Equations (2 – 4), and our prior work also implemented and compared four different modulus and three modular reduction methods [24]. The experimental results shows that if the Solinas modulus [27], $2^{64} - 2^{32} + 1$, is used, the super-size multiplier consumes the shortest multipli-

cation time. Considering the hardware cost, although the Solinas modulus is a little more complex than the special modulus form $p = 2^{m-1} + 1$, it is much cheaper than the Barrett reduction [28] as two on-line multiplications are required for the Barrett reduction. So in this paper, we continue to adopt the Solinas modulus. This means the parameters listed in Section 2 are assigned with values as: $p = 2^{64} - 2^{32} + 1$, $m = 64$, $\omega = 7$ and $b \leq 28$. The base bit-length, b, determines the valid data processing rate, that is, how many bits of the useful data are processed when we perform the 64-bit modular p arithmetic. Therefore, we choose the largest value for b in our design, i.e., $b = 28$.

The modular reduction architecture of the Solinas modulus is shown in Fig. 10. It is also identical to the prior work [14]. Interested reader can learn the details in [24].

3.5 The Addition-Recovery and Product-Accumulation Module

The addition-recovery module is responsible for converting the IFFT outputs back to an integer by resolving a very long carry chain, as is shown in Equation (5). The product-accumulation module is used to combine the block products, which are shown in Fig. 2(ii), to form the final multiplication results that can be written to memory. As they are tightly coupled together in our design, they are described in the same section. As valid output count of the FFT/IFFT module is changed into 2 from k in the previous design [24], the addition recovery and product accumulation module is completely re-designed, and is illustrated in Fig. 11. Readers can understand the logic in it by combining Fig. 2(ii), Fig. 8 and Fig. 11.

Observing the IFFT output data index pattern illustrated in Fig. 8, it can be seen that the LSB_half and MSB_half results of a block product is successively output pair by pair. Thus, the architecture in Fig. 11 is composed of two parts. The up and bottom parts are respectively responsible for computing the LSB_half and MSB_half result of a block product, and as illustrated in the block product $x_0 \times y_0$ in Fig. 2. As there are at most 4 inputs for an addition operation, so the bit width of the addition result is equal to $m + 2$. As the base unit bit-length is equal to b, the write/read bus bit-width of RAM access is also equal to b. The $m + 2 - b$ carry bits are kept in the on-chip register array. Since there are parallel two adders, there are also 2 RAM access ports. In order to achieve a pipeline design, each RAM port has both of independent b-bit write bus and read bus.

As displayed in Fig 11, there are 4 cases for the LSB_half block product addition operation. The first case is that the m-bit Up-Branch output of the IFFT module is directly assigned to the LSB_half block product addition result. This case occurs when the first b-bit result of the first LSB_half block product, $x_0 \times y_0$, is generated. The 2nd case is that the Up-Branch output is added to the $(m + 2 - b)$-bit MSB addition result generated in the previous clock cycle. The case occurs when the subsequent b-bit results of $x_0 \times y_0$, and of all the other LSB_half products, for example $x_1 \times y_0$ or $x_2 \times y_0$, are generated. So both of the 1st and 2nd cases only involve the function of addition-recovery, and these two cases are used in the first outer-iteration, as the outer-iteration-0 situation illustrated in Fig. 2. The 3rd case has three inputs: the Up-Branch output, the MSB addition result of the previous clock, and the b-bit RAM

result of the previous block accumulation products. This case occurs when the block products of outer-iteration-1, $x_0 \times y_1$ or $x_1 \times y_1$, are generated. And the corresponding previous block products are $x_1 \times y_0$ or $x_2 \times y_0$ during the outer-iteration-0. The 4th case requires four inputs: the Up-Branch output, the MSB addition result of the previous clock, the RAM result of the previous block accumulation products, and the $(m + 2 - b)$-bit MSB addition result of the previous block product, which is illustrated as the $(m + 2 - b)$-bit carry bits in Fig. 11. This case only occurs at the moment of the first clock cycle of each block product in the 2nd or subsequent out-iteration, for example $x_0 \times y_1$ to $x_4 \times y_1$ in the outer-iteration-1 in Fig. 2. The register of carry bits stores the final $(m + 2 - b)$-bit LSB_half MSB addition result of $x_0 \times y_0$ to $x_4 \times y_0$. So the 3rd and 4th cases involve both of the addition-recovery and product-accumulation function, and they are only used when more than a single outer-iteration is needed.

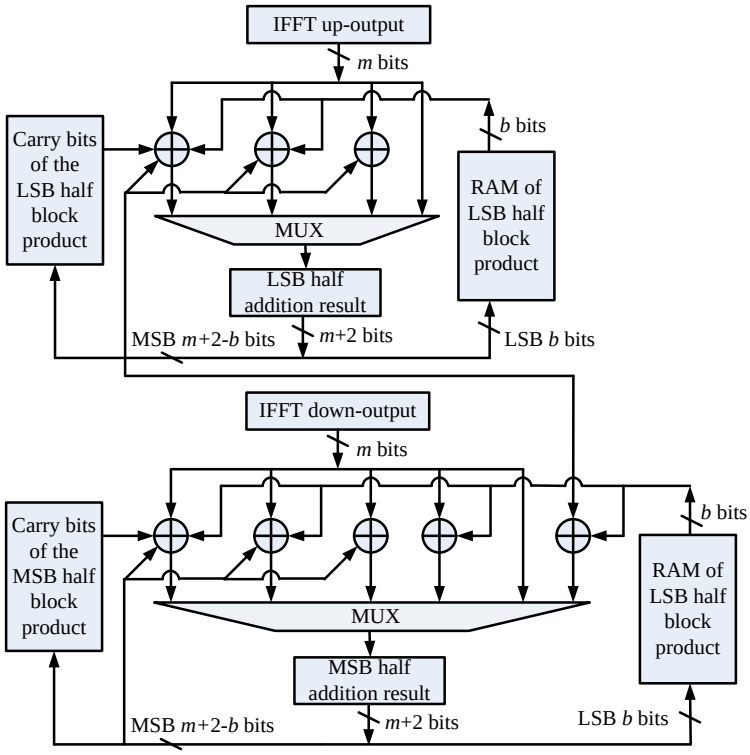


Fig. 11. The proposed addition-recovery and block-accumulation module used in the proposed basic architecture.

In Fig. 11, there are 6 cases for the addition operation of the MSB_half block product. The first case is that the m-bit Down-Branch output of the IFFT module is directly assigned to the addition result of the MSB_half block product. The 2nd case is that the Down-Branch output is added to the $(m + 2 - b)$-bit MSB addition result of

the MSB_half block product generated in the previous clock cycle. The 3rd case needs three inputs: the Down-Branch output, the previous clock MSB addition result, and the b-bit RAM result of the previous block accumulation products. The 4th case requires four inputs: the Down-Branch output, the MSB addition result of the previous clock, the previous block accumulation products RAM result, and the $(m+2-b)$-bit MSB addition result of the previous MSB_half block product, which is also illustrated as the $(m+2-b)$-bit carry bits in Fig. 11. All these four cases share the same operation conditions of the LSB_half block product.

It can be found that the four addition cases of the MSB_half and LSB_half block product can generate the correct product-accumulation result from $x_0 \times y_0$ to the LSB_half of $x_4 \times y_1$, however, it cannot add the final $(m+2-b)$-bit MSB addition result of the LSB_half of $x_4 \times y_0$ to the beginning bits of the addition result of the MSB_half of $x_4 \times y_1$. Thus, the 5th and 6th cases are used to deal with this situation. The 5th case adds the final $(m+2-b)$-bit MSB addition result of the LSB_half of $x_4 \times y_0$ to the b-bit RAM result of the current MSB_half of $x_4 \times y_1$. The 6th case is used to add the the $(m+2-b)$-bit MSB addition result of the previous clock to the b-bit RAM result of the current MSB_half of $x_4 \times y_1$. It can also find that these two cases only occurred at the final stage of the product-accumulation.

In the addition-recovery and product-accumulation module, two adders are used as displayed in Fig. 11, and each adder has its separate read and write port, and each port is b-bit. So the total bit-width of the RAM read and write ports is $4b$. As both the write address range of the two adders and the read address range of the two adders are equal to the whole range of the super-size multiplication product, so the total bit-width of the RAM write and read address buses is equal to $4\log_2\frac{n_x+n_y}{b}$, where we assume the input operand is x and y, and their bit-length is n_x and n_y.

Above all, including the port bit-width in Section 3.2, the total data and address bus bit-width of the basic architecture is $6b + \log_2\frac{n_x}{b} + \log_2\frac{n_y}{b} + 4\log_2\frac{n_x+n_y}{b}$.

3.6 Time Latency of The Proposed Basic Architecture

For a k-point Interger-FFT algorithm, each FFT/IFFT module has $\log_2 k$ processing stages. Each processing stage contains a butterfly module. Let the count of pipeline stage in a FFT butterfly be N_F, and the count of the pipeline stages in an IFFT butterfly be N_{IF}. Then the latency of all these FFT and IFFT butterfly modules can be computed as $(N_F + N_{IF})(\log_2 k - 1) + 2$, where we assume the addition only takes 1 clock cycle, so the total latency of the two butterflies in the 1st processing stage of FFT and IFFT is equal to 2. As the buffer count of the FFT module is $k/4$, $k/8$, …, until 1, and the total buffer count between the FFT and the IFFT is the same, the latency of all the buffers in the FFT and IFFT modules are computed as: $2\left(\sum_{i=2}^{\log_2 k} k/2^i\right)$. Let the pipeline stage count of the point-wise multiplication module be N_{PW}, then the latency of generating the first $2b$-bit IFFT result (one b-bit is for the LSB_half addition result and is a valid multiplication result, the other b-bit is for the MSB_half addition result and would be accumulated again in the later) can be computed as:

$$\Delta_0 = 2\left(\sum_{i=2}^{\log_2 k} \frac{k}{2^i}\right) + (N_F + N_{IF})(\log_2 k - 1) + 2 + N_{PW} \quad (6)$$

where N_F, N_{IF}, N_{PW} are implementation-related.

As our proposed design is a pipeline design (i.e., all the FFT, IFFT, addition recovery, product accumulation modules are pipelined), after the first $2b$-bit IFFT result is generated, $2b$-bit IFFT result is generated in each subsequent clock. As we use a k-point IFFT, each inner-iteration (i.e., each block product generation) requires $k/2$ clock cycles. Assuming we perform a super-size multiplication, $z = x \times y$, let n_x and n_y be the bit length of the super-size operands x and y respectively, then, the total count of inner-iteration in this situation is equal to $\left\lceil \frac{n_x}{(kb)/2} \right\rceil \left\lceil \frac{n_y}{(kb)/2} \right\rceil$, where $kb/2$ is equal to the used data bit-length of each operand in a single block product computation. According to the description in Section 3.5, the first four cases of MSB_half and LSB_half addition are completed in $k/2$ clock cycles. The 5th and 6th cases need an additional $k/2$ clock cycles to complete the final product accumulation. So the total clock cycles of the block product accumulation is equal to

$$\Delta_1 = \left(\left\lceil \frac{n_x}{(kb)/2} \right\rceil \left\lceil \frac{n_y}{(kb)/2} \right\rceil + 1\right)\frac{k}{2} \quad (7)$$

Then the total latency of $z = x \times y$ using the proposed basic architecture can be estimated as

$$\Delta = \Delta_0 + \Delta_1 \quad (8)$$

4 The Proposed Modified Multiplier Architecture

In this section, we propose an improvement of the proposed basic architecture in Section 3. The aim is to reduce the latency. We try to make the latency of the proposed architecture as small as possible, with the constraint that the proposed architecture does not exceed the hardware resource budget of the targeted FPGA platform. The top-level overview diagram of the proposed modified architecture is the same as that illustrated in Fig. 1. The FFT, IFFT and point-wise multiplication modules are identical to the modules in Section 3, so no further description is given any more.

4.1 The Modified Integer-FFT and FSM Controller Architecture

The core idea is to increase the parallel processing ability of the basic design in Fig. 2 to tradeoff a decreased latency. The proposed modified Integer-FFT architecture and the FSM controller logic is illustrated in Fig. 12. Compared to the basic design in Fig. 2, one more IFFT module and one more point-wise multiplication module are added in this modified architecture.

The reason for the change in Fig. 12(i) can be explained by the FSM controller logic illustrated in Fig. 12(ii). The count of block products in Fig. 12(ii) is the same as that in Fig. 2(ii). The outer-iteration count is also the same as the basic design. The inner-iteration-0 remains unchanged. What does change is the subsequent inner-

iteration method. As we instantiate two IFFT modules, two block products can be parallel processed in one inner-iteration, for example, $x_1 \times y_0$ and $x_2 \times y_0$ in Fig. 12(ii). In our previous design in Fig. 2(i), the right FFT module serially processes the blocks of x, and the left FFT module always deals with the same data of y in each outer-iteration, for example, in the outer-iteration-0 it repeatedly 5 times processes the same y_0 in Fig. 2(ii). Thus, it is actually a resource waste. In the modified design in Fig. 12(ii), the operation of both the two FFT modules is the same as that of the basic design in the inner-iteration-0, however, the obtained value Y_0 is written into the off-chip RAM when the block product $x_0 \times y_0$ is generated in the modified design. Then at the following inner-iterations, both of the two FFT modules can start to deal with the blocks of x pair by pair. For example in Fig. 12(ii), the right FFT module starts to process x_1 and x_3, and the left FFT module can be used to deal with the data of x_2, x_4. Thus, the inner-iteration count is reduced to 3 in Fig. 12(ii), rather than 5 in Fig. 2(ii). As the outer-iteration count is not changed, when the block count of x is much greater than 1, the total inner-iteration count can be reduced to almost a half of the basic design asymptotically, thus, the total latency of the modified design can be reduced up to a half of the basic design asymptotically. Compared to our prior architecture [24], this modified architecture saves a lot of hardware resources. Specifically, the butterfly count of all FFT and IFFT modules is reduced from $3 \times k/2 \times \log_2 k$ to $4 \times \log_2 k$, and the point-wise multiplication modules is reduced from k to 4.

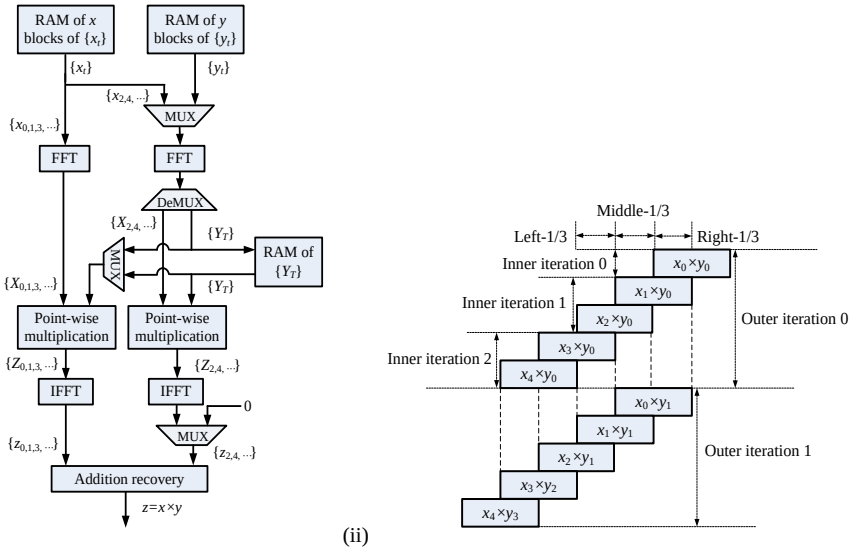


Fig. 12. (i) The proposed Integer-FFT module architecture used in the proposed modified architecture. (ii) The proposed block-accumulation logic used in the proposed modified architecture.

In the modified design, we have two FFT modules unchanged, so their data bus and address bus bit-width stay the same as before, $2b + \log_2 \frac{n_x}{b} + \log_2 \frac{n_y}{b}$.

4.2 The Addition-Recovery and Product-Accumulation Module

As one more IFFT module is added and two block products are generated in each inner-iteration in the modified architecture, the addition-recovery and product-accumulation module is re-designed and illustrated in Fig. 13. Readers can understand the logic in them by combining them with Fig. 8 and Fig. 12.

As the valid output count of the IFFT module is changed into 4 from 2 in the basic design, three adders are needed. For example, in the inner-iteration-1, the block products $x_1 \times y_0$ and $x_2 \times y_0$ are generated in parallel, and the position of the MSB_half $x_1 \times y_0$ overlaps that of the LSB_half $x_2 \times y_0$. Thus, one adder is used for computing the LSB_half of $x_1 \times y_0$ (i.e., Right_1/3 in Fig. 12(ii)), the 2nd adder is used for adding the MSB_half $x_1 \times y_0$ and the LSB_half $x_2 \times y_0$ (i.e., Middle_1/3 in Fig. 12(ii)), and the 3rd adder is used for the MSB_half $x_2 \times y_0$ (i.e., Left_1/3 in Fig. 12(ii)). Their computational architecture is illustrated in Fig. 13(i), and is composed of three additions. The up, middle and bottom parts are respectively responsible for computing the Right_1/3, Middle_1/3 and MSB_1/3 result, and their position example is illustrated in Fig. 12(ii). As there are at most 5 inputs for an addition operation, so the bit-width of the addition result is equal to $m + 3$. As the base unit bit-length is equal to b, the bit-width of RAM access (write/read) is also equal to b. The other $m + 3 - b$ carry bits are kept in the on-chip register array. In order to achieve a pipeline design, each RAM access port has both of the b-bit write bus and the read bus. So the total bit-width of the RAM read and write data ports is equal to $6b$. As the write/read address range of the 3 adders is equal to the whole range of the super-size multiplication product, so the total bit-width of write and read address bus is equal to $6 \log_2 \frac{n_x+n_y}{b}$.

When the count of inner-iterations is less than or equal to 2, it can be found that the logic in Fig. 12(ii) is the same as that in Fig. 2(ii). Thus, the first 4 cases of the 1st adder and the first 6 cases of the 2nd adder are the same as that in Fig. 12. The other cases are also easier to understand according to the Fig. 12(ii), and have the similar logic as described in Section 3.5. Thus, they are not detailed here. The situation of the 3rd adder for the Left-1/3 addition result is similar to the MSB_half addition result in Fig. 11. The only exception is that the absence of the input from the MSB Middle_1/3 result like the input from the MSB MSB_half result in Fig. 11. The reason is that the addition advance step is $2b$-bit, so the MSB Middle_1/3 result is already added into the Right_1/3 result as described before.

However, it can be it can be found that the LSB_half $x_3 \times y_0$ is aligned with the MSB_half $x_2 \times y_0$, so the final carry bits of the LSB_half $x_1 \times y_0$ cannot be accumulated into the product due to the $k/2$ distance. The essence is that the product is generated with the $2d$-bit advance step, thus, the product-accumulation function in these three adders cannot catch up with the speed of the $2d$-bit advance step. Therefore, the 4th adder is required in the modified architecture with the $2d$-bit width RAM read and write buses. This is illustrated in Fig. 13(ii). It must be noted that this module only starts to function when the x's data block count is equal to or greater than 3. And it starts to work in $k/2$ clocks lagging behind the other 3 adders, due to the fact that its advancing step speed is $2d$-bit. So the correct accumulation pipeline can be formed. For example, in Fig. 12(ii), when the first 3 adders work on the block products

$x_3 \times y_0$ and $x_4 \times y_0$ using the d-bit step of $k/2$ clocks, and their position coverage range is $1\frac{1}{2}$ block products, the 4th adder works on the whole block product $x_1 \times y_0$ and LSB_half $x_2 \times y_0$ using the $2d$-bit step of $k/2$ clocks, and its position coverage range is 1 block product. Therefore, the scheme guarantees that the advancing distance between the first 3 adders and the 4th adder is equal to ½ block product. Finally, when the count of x's data block is even, $k/4$ clocks is needed for the 4th adder to cover the final one block product result after the first 3 adders finish their job; when the count of x's data block is odd, $3k/4$ clocks is needed for the 4th adder to cover the final two block product results after the first 3 adders finish.

The 4th adder has its separate $2b$-bit read and write buses. Its write and read address range is also equal to the whole range of the super-size multiplication product. Thus, the total data and address bus bit-width of the modified architecture is $12b + \log_2\frac{n_x}{b} + \log_2\frac{n_y}{b} + 6\log_2\frac{n_x+n_y}{b} + 2\log_2\frac{n_x+n_y}{2b}$.

4.3 Time Latency of The Proposed Modified Architecture

Using the same analysis assumption and method of Section 3.6, the time latency of the modified architecture can be obtained. As the first inner-iteration in the two architecture is the same, the latency of generating the first $3b$-bit IFFT result (one b-bit is for the Right_1/3 addition result and is a valid multiplication result, the other $2b$-bit is for the Middle_1/3 and Left_1/3 addition results and would be accumulated again in the later) is the same as before, Δ_0.

When the x's data block count is equal to or less than 2, the total count of inner-iteration is the same as the basic design. Otherwise, the total count of inner-iteration in this situation is equal to $(\left\lceil\frac{n_x}{(kb)/2}\right\rceil - 1)/2 \times \left\lceil\frac{n_y}{(kb)/2}\right\rceil$. So the total clock cycles of the first 3 adders is equal to

$$\Delta_2 = \begin{cases} \left(\left\lceil\frac{n_x}{(kb)/2}\right\rceil\left\lceil\frac{n_y}{(kb)/2}\right\rceil + 1\right)\frac{k}{2}, when\ x\ data\ block\ count \le 2 \\ \left(\left\lceil\left(\left\lceil\frac{n_x}{(kb)/2}\right\rceil - 1\right)/2\right\rceil \times \left\lceil\frac{n_y}{(kb)/2}\right\rceil\right)\frac{k}{2}, otherwise \end{cases} \quad (9)$$

The latency of the 4th adder for the final product-accumulation is

$$\Delta_3 = \begin{cases} k/4, when\ x\ data\ block\ count\ is\ even \\ 3k/4, when\ x'sdata\ block\ count\ is\ odd \end{cases} \quad (10)$$

Then the total latency of $z = x \times y$ using the proposed modified architecture can be estimated as:

$$\Delta' = \Delta_0 + \Delta_2 + \Delta_3 \quad (11)$$

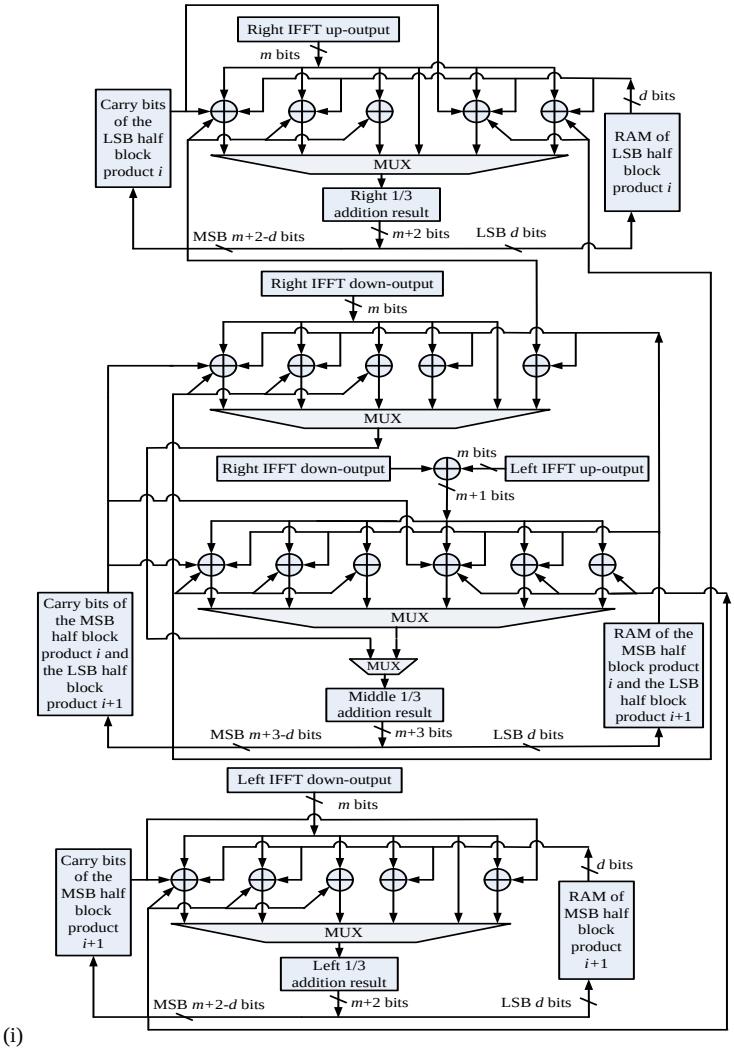


Fig. 13. The proposed addition-recovery and product-accumulation module used in the proposed modified architecture.

5 Implementation, Performance and Comparison

All the proposed architectures were designed and verified using Verilog language and Xilinx FPGA. Modelsim 6.5a was used as the functional and post-synthesis timing simulation tool. The synthesis tool used was Xilinx ISE Design Suite 14.4. The synthesis strategy is set to balance between speed and area. The optimization objective is set to speed. The target device is Virtex-7 XC7VX980T. The test vectors are generated as random numbers in C language with Visual Studio.

5.1 Implementation and Performance

The proposed two architectures are instantiated by 256, 512, 1024, 2048, and 8192-point FFT respectively. In our implementations, for the small size 64-bit×64-bit multiplication used in FFT butterfly and point-wise multiplication, Xilinx Core Generator is employed to automatically generate a 12 stage pipeline multiplier using the embedded multipliers resource, DSP48E1. And the pipeline stage count is determined by try-and-error method. Then, $N_F = N_{IF}$ is designed to be 17 and N_{PW} is designed to 15 in all the implementations. We also set $n_x = n_y = b \times 2^b$, so the bit-length of both input operands is more than 1 gigabits, which is sufficient for our simulation and verification. Then the total bit width of the data and address buses in the basic and modified architecture is equal to 340 and 620.

Table 1. The synthesis results of the proposed super-size multipliers with two architectures.

	FFT Point	**Frequency (MHz)**	**Slice Register Utilisation (total: 1224000)**	**Slice LUT Utilisation (total: 612000)**	**DSP48E1 Utilisation (total: 3600)**	**RAM access bit width**
The basic architecture	256	245.408	28406	36250	384	340
	512	245.408	32136	41917	432	340
	1024	245.408	35849	48778	480	340
	2048	252.950	39814	58125	528	340
	4096	252.950	43727	72484	576	340
	8192	243.792	47772	97178	624	340
The modified architecture	256	161.276	39643	54068	544	622
	512	161.276	44599	61668	608	622
	1024	161.276	49538	71060	672	622
	2048	161.453	54795	83930	736	622
	4096	161.453	59996	103909	800	622
	8192	159.499	64765	138175	864	622

The synthesis results of the proposed implementations using the basic and modified architecture are displayed in Table 1. It can be seen that the proposed 8192-point implementation using the basic architecture in this paper is within the hardware resource budget of the Virtex-7 XC7VX980T: the Slice Register count is no more than 4% of the available resource, Slice LUT count is about 16%, and the DSP48E1 is only 18%, the data and address bus bit width is no more than 50%. The 8192-point FFT implementation of the modified architecture also is within the budget: the Slice Register

count is only 5%, Slice LUT is no more than 22%, the DSP48E1 is about 24%, and the RAM access bit width is only about 87%. It can also be found that the synthesis frequency of the modified implementations is worse than that of the basic implementations. The reason is that the more complicated controller logic and the increased addition logic level, which can be seen from the Fig. 11 and Fig. 13.

Table 2. The latency (clock cycle count) of the proposed architectures for an integer multiplication with the same bit-length operands.

		Bit-length of both operands						
	FFT point	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}
The basic architecture	256	1147	1787	3835	13435	46843	175867	701563
	512	1309	2077	3357	7453	26653	93469	351517
	1024	2367	2367	3903	6463	14655	53055	186687
	2048	4449	4449	4449	7521	12641	29025	105825
	4096	8579	8579	8579	8579	14723	24963	57731
	8192	16805	16805	16805	16805	16805	29093	49573
The modified architecture	256	1019	1339	2363	7163	23867	88379	351227
	512	1437	1821	2461	4509	14109	47517	176541
	1024	2623	2623	3391	4671	8767	27967	94783
	2048	4961	4961	4961	6497	9057	17249	55649
	4096	9603	9603	9603	9603	12675	17795	34179
	8192	18853	18853	18853	18853	18853	24997	35237
		Bit-length of both operands						
	FFT point	2^{19} ($\times 10^3$)	2^{20} ($\times 10^3$)	2^{21} ($\times 10^4$)	2^{22} ($\times 10^4$)	2^{23} ($\times 10^5$)	2^{24} ($\times 10^5$)	2^{25} ($\times 10^6$)
The basic architecture	256	2766.6	10989	4395.5	17552	7014.8	28059	11221
	512	1402.9	5533.0	2197.8	8791.0	3510.4	14030	5611.8
	1024	702.78	2805.6	1106.6	4395.7	1758.2	7020.8	2805.9
	2048	373.09	1405.3	561.08	2213.1	879.13	3516.4	1404.2
	4096	211.33	745.86	281.02	1122.1	442.62	1758.3	703.28
	8192	115.11	422.31	149.14	562.01	224.42	885.23	351.65
The modified architecture	256	1383.7	5495.1	2197.8	8776.0	3507.4	14030	5610.6
	512	702.24	2767.3	1099.0	4395.6	1755.2	7014.8	2805.9
	1024	352.83	1404.2	553.43	2198.0	879.12	3510.4	1403.0
	2048	189.28	705.38	280.82	1106.8	439.59	1758.2	702.08
	4096	110.98	378.24	141.04	561.60	221.36	879.18	351.65
	8192	680.05	221.61	75.613	282.05	112.31	442.72	175.84

($\times 10^3$) is used as the unit of the clock cycle count in the same column, and the others works for the same purpose.

Using the parameter in the implementations, the estimated multiplication latency with the same bit-length operands of the basic and modified architecture is illustrated in Table 2. From Table 2, it can be seen that the latency analysis in Equation (8) and Equation (11) is verified, that is, the latency of the modified implementation should approximate ½ of the basic design with the increase of the operand bit-length asymptotically. It can also be seen that different FFT versions have their different and best suitable situations. When the bit-length of the operand is much small, for example, 2^{12} or 2^{13}, the 256 or 512-point version is the best; when the bit-length of the operand is more than 2^{18}, the 4096 or 8192-point version outperforms the other implementations; when the bit-length of the operand is more than 2^{20}, the latency of the next

larger point FFT becomes a half of the current FFT. Both of the two architectures share the same characteristic and trend.

Table 3. The running time (in milliseconds) of the proposed architectures running of an integer multiplication with the same bit-length operands.

	FFT point	**Bit-length of both operands**						
		2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}
The basic architecture	256	0.005	0.007	0.016	0.055	0.191	0.717	2.859
	512	0.005	0.008	0.014	0.030	0.109	0.381	1.432
	1024	0.010	0.010	0.016	0.026	0.060	0.216	0.761
	2048	0.018	0.018	0.018	0.030	0.050	0.115	0.418
	4096	0.034	0.034	0.034	0.034	0.059	0.099	0.228
	8192	0.069	0.069	0.069	0.069	0.069	0.119	0.203
The modified architecture	256	0.006	0.008	0.015	0.044	0.148	0.548	2.178
	512	0.009	0.011	0.015	0.028	0.087	0.295	1.095
	1024	0.016	0.016	0.021	0.029	0.054	0.173	0.588
	2048	0.031	0.031	0.031	0.040	0.056	0.107	0.345
	4096	0.059	0.059	0.059	0.059	0.079	0.110	0.212
	8192	0.118	0.118	0.118	0.118	0.118	0.157	0.221
	FFT point	**Bit-length of both operands**						
		2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}
The basic architecture	256	11.273	44.780	179.111	715.215	2858.410	11433.631	45724.748
	512	5.717	22.546	89.559	358.221	1430.429	5716.818	22867.261
	1024	2.864	11.432	45.091	179.116	716.442	2860.857	11433.636
	2048	1.475	5.556	22.182	87.492	347.550	1390.160	5551.114
	4096	0.835	2.949	11.110	44.362	174.982	695.099	2780.318
	8192	0.472	1.732	6.117	23.053	92.056	363.109	1442.419
The modified architecture	256	8.580	34.073	136.276	544.161	2174.773	8699.079	34788.871
	512	4.354	17.159	68.144	272.551	1088.322	4349.545	17398.157
	1024	2.188	8.707	34.316	136.286	545.101	2176.642	8699.089
	2048	1.172	4.369	17.393	68.554	272.272	1089.005	4348.509
	4096	0.687	2.343	8.736	34.784	137.106	544.542	2178.009
	8192	0.426	1.389	4.741	17.684	70.418	277.569	1102.424

The experimental multiplication time is shown in Table 3. From Table 3, it can be found that the running time of the modified architecture is about 2/3 rather than ½ of the basic architecture. The reason is that the synthesis frequency of the implementations of the modified architecture is less than that of the basic architecture implementations listed in Table 1.

5.2 Evaluating CNT's FHE Multiplication

As the experiments show the modified architecture outperforms the basic architecture, we use the modified implementations to evaluate the super-size multiplications used in the encryption primitive of one scheme of FHE over the integers, which is presented in 2012 by Coron *et al.* [5], and it is denoted as CNT here. The definition of the CNT encryption primitive is in Equation (12).

$$C = \left(M + 2R + 2\sum_{i=1}^{\theta} B_i \times A_i\right) mod\ A_0 \tag{12}$$

where C denotes the ciphertext; $M \in \{0,1\}$ is a 1-bit plaintext; R is a random signed integer in $(-2^{\rho}, 2^{\rho})$; $A_0 \in [0, 2^{\varphi})$, is a part of the public key. In Equation (12), $\{B_i\}$ with $1 \leq i \leq \theta$ is a random integer sequence, and each B_i is a δ-bit integer; $\{A_i\}$ with $1 \leq i \leq \theta$ is the public key sequences, and each A_i is a φ-bit integer. In the original paper, the four test group is given in Table 4.

From Equation (12), we know there are two places requiring multiplications, one for $B_i \times A_i$, and the other for $(\cdot)\ mod\ A_0$. If we adopt the super-size Barrett reduction idea in our prior work [24], we know each Barrett reduction requires two multiplications with the input operand bit-length, $\delta + \theta$ and φ. And according to the parameters in CNT [5], we can give out the required four operand bit-length for $B_i \times A_i$ and $(\cdot)\ mod\ A_0$ in Table 5.

Table 4. The four test groups of parameters for Equation (12) in CNT [5]

Group	δ	$\rho = \delta$	$\varphi \times 10^{-6}$	θ
Toy	936	936	0.15	158
Small	1476	1476	0.83	572
Medium	2016	2016	4.20	2110
Large	2556	2556	19.35	7695

Table 5. The multiplication operand bit-length required in Equation (12)

Group	1st operand $(\delta + \theta)$	2nd operand $(\varphi \times 10^{-6})$	Group	1st operand $(\delta + \theta)$	2nd operand $(\varphi \times 10^{-6})$
Toy I	936	0.15	Toy I	1094	0.15
Small I	1476	0.83	Small I	2048	0.83
Medium I	2016	4.20	Medium I	2226	4.20
Large I	2556	19.35	Large I	10251	19.35

Table 6. The latency (clock cycle count) of an CNT multiplication with the proposed modified architecture.

	Toy I	Small I	Medium I	Large I	Toy II	Small II	Medium II	Large II
256	3451	15611	75771	346299	3451	15611	150779	1037371
512	3997	16157	76317	346909	3997	16157	76317	692509
1024	5183	17215	77375	347967	5183	17215	77375	347967
2048	7521	19297	79713	350049	7521	19297	79713	350049
4096	11651	23939	84355	354691	11651	23939	84355	354691
8192	20901	33189	92581	362917	20901	33189	92581	362917

We note the property of the multiplication in CNT is unbalanced input operand bit-length, unlike the same bit-length operand used in Table 2 and Table 3. So we must carefully compare and choose the suitable FFT point count. The Table 6 and Table 7 give out the required latency and simulated running time of CNT FHE multiplications and compares them. It can be observed that the latency changing trend of the CNT multiplication is different from the multiplication with the same bit-length operands. The finding is that a larger point FFT does not represent a better performance. For the Medium-II and Large-II groups, 1024-point or larger point FFT should be the best

choice as they consume the least time; for the other groups, the 256-point or 512-point FFT implementations should be the best candidates.

Table 7. The time (in milliseconds) of an CNT multiplication with the proposed modified architecture.

	Toy I	Small I	Medium I	Large I	Toy II	Small II	Medium II	Large II
256	0.021	0.097	0.470	2.147	0.021	0.097	0.935	6.432
512	0.025	0.100	0.473	2.151	0.025	0.100	0.473	4.294
1024	0.032	0.107	0.480	2.156	0.032	0.107	0.480	2.158
2048	0.047	0.120	0.494	2.168	0.047	0.120	0.494	2.168
4096	0.072	0.148	0.522	2.197	0.072	0.148	0.522	2.197
8192	0.131	0.208	0.580	2.275	0.131	0.208	0.580	2.275

Table 7 has some differences compared to Table 6. This is caused by the different synthesis frequency. However, their trend is the same. It can be seen that different group needs different FFT point count in order to minimise the running time. Therefore, according to Table 6 and Table 7, we can make the choice of which version of the modified implementations should be used to implement which group of the CNT FHE encryption primitives.

Here we adopt the FHE encryption implementation logic in our prior work [24]. It is assumed there is only one hardware multiplier, and the accumualtion operation $\sum_{i=1}^{\theta} B_i \times A_i$ and the Barrett reduction operation $(\cdot)\ mod\ A_0$ serially call the same multiplier, then the total running time can be determined by the sum of accumulation time and Barrett reductiont time. The accumulation time is equal to the product of a single multiplication time of Type-I, which is listed in Table 7, and accumulation iteration count, which is listed in Table 4. The Barrett reduction time is equal to the twice multiplication time of Type-II, which is listed in Table 7. Therefore, the estimated CNT encryption primtive running time is given in Table 8. For example, the Toy group shoud be implemented by 256-point FFT implementation, and its single multiplication time of Type-I is equal to 0.021 ms, and its single multiplication time of Type-II is 0.021 ms, so the encryption time is $0.021 \times 158 + 0.021 \times 2 = 3.36$ ms.

Table 8. The estimated time (in milliseconds) of an CNT encryption with the proposed modified architecture.

	The proposed modifed implementations	One encryption time (milliseconds)
CNT Toy	256-point FFT	3.36
CNT Small	256-point FFT	55.561
CNT Medium	512-point FFT	999.414
CNT Large	1024-point FFT	16606.950

5.3 Comparison

In this section, we compare the proposed performance of our modified implementations to the previous literatures.

Table 9. The comparison of multiplicaion time (in milliseconds) with the same operand bit-length.

	FFT Point	Platform	Bit-length of both operands			
			2^{21}	2^{22}	2^{23}	2^{24}
This work	8192	Virtex-7 FPGA	4.741	17.684	70.418	277.569
[18]	2048K	Core 2 Q6600	23.576	53.653	141.655	297.032
	2048K	Core i7 870	15.738	35.283	80.479	186.751
	2048K	GTX 295	3.544	7.968	16.627	27.841
	2048K	GTX 480	1.122	2.395	4.924	9.666

The running time of the same operand bit-length multiplications is compared in Table 9. It can be seen that the proposed 8192-point implementation outperforms the software implementation on Core 2 Q6600 platform, and when the bit-length is less than or equal to 2^{23}, it also is better than the software implementation on Core i7 870 platform. However, the performance advantage becomes less and less with the increasing of the bit-length. Another observation is that it works poorer than the same software implementation on the two GPU platforms, GTX 295 and GTX 480. The reason is that the software implementation uses a very large 2048K-point FFT and the GPU's multi-core parallel computing technology. However, it must be noted that the latency changing trend illustrated in Table 3 of our proposed architecture: the next larger point count FFT will only require ½ of the latency required by the current FFT version. So if a (8192×32)-point FFT implementation of our proposed modified architecture is used, it is believed that its latency will be 1/32 of the 8192-point version, and it will outperform the software implementation on GTX 480.

The hardware resource cost comparison result is given in Table 10, where we list our largest version, 8192-point FFT implementation and our prior work. As the prior work requires at least four read ports of the same bit-width and each of them is 224 [24], its total bit-width of the data and address buses is greater than 896. It can be seen that the proposed 8192-point implementation using the modified architecture in this paper saves a lot of hardware resource compared to the prior work: the Slice Register count is no more than 1/17 of the prior work, Slice LUT count is about 1/7, and the DSP48E1 is only 1/21, the data and address bus bit-width is no more than 2/3.

Table 10. The comparison of hardware resource cost.

	FFT Point	Frequency (MHz)	Slice Register Utilisation (total: 1224000)	Slice LUT Utilisation (total: 612000)	DSP48E1 Utilisation (total: 3600)	RAM access bit width
This work	8192	159.499	64765	138175	864	622
[24]	128	166.450	1122826	954737	18496	>224*4=896

The CNT FHE encryption primitive running time comparison result is given in Table 11. Compared to the corresponding CNT software implementation [5] on the Intel Core2 Duo E8400, It can be seen that the proposed implementations are almost 14.6 times quicker than CNT for the Toy group, also achieve 26.2 times improvement for the Large group. Compared to the work implemented on a GPU platform [20], our

work also runs 30 times quicker. It also shows that our proposed work requires more time than our prior work [24]. The reason is that the prior work uses a fully parallel FFT architecture, so the valid multiplication product bit-length generated in each clock is much larger than this work. Specifically, in our work 56 bits are generated each clock, but the prior work generates 224 bits in each clock. However, our current work in this paper consumes much less hardware cost than our prior work, and the prior work obviously exceeds too much beyond the available Virtex-7 hardware resource.

Table 11. The average running time comparison of the proposed FHE encryption designs.

Group	Toy	Small	Medium	Large
This work	0.003424 s	0.05556s	0.9994s	16.607 s
[24]	0.000739 s	0.0132 s	0.4772 s	7.994 s
[5]	0.05 s	1.0 s	21 s	7 min 15 s
[20]		1.69 s		

6 Conclusion

In this paper, the novel hardware architectures and implementations of super-size integer multiplications on Xilinx Virtex-7 FPGA are proposed. First of all, the serial FFT architecture is utilised to propose a basic Integer-FFT multiplication architecture with the feature of lower hardware cost. Then, in order to accelerate the multiplication speed within the hardware cost budget, a modified Integer-FFT architecture is proposed. Next, both architectures are implemented and verified on the Virtex-7 FPGA platform using 256, 512, 1024, 2048 and 8192 point Integer-FFT algorithm respectively with multiplication operand bit-length ranging from 2^{12} bits to 2^{25} bits. Experimental results show that the hardware cost of the proposed architecture requires only about 1/17 of Slice Registers, 1/7 of Slice LUT, 1/21 of DSP48E1 and 2/3 IO resource of the prior work. If the same bit-length operand is used, the proposed implementation outperforms the corresponding software implementations on the Core 2 Q6600 and Core i7 870 platforms. When the proposed implementations are employed to evaluate the super-size multiplication in an integer-based FHE encryption primitive, the result shows that the proposed work runs up to 26.2 times faster than the corresponding software on the Core2 Duo E8400 platform. In summary, the proposed implementation achieves a better practical purpose than the prior work, as it not only accelerates the speed of the corresponding FHE encryption primitive, but also saves a lot of hardware resource.

References

1. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University, (2009). Http://crypto.stanford.edu/craig.
2. Gentry, C.: Fully homomorphic encryption using ideal lattices. The 41st annual ACM symposium on Theory of computing. pp. 169–178. ACM (2009).

3. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. EUROCRYPT 2010, LNCS, vol. 6110, pp. 24–43. Springer (2010).
4. Coron, J.S., Mandal, A., Naccache, D., Tibouchi, M: Fully Homomorphic Encryption over the Integers with Shorter Public Keys. CRYPTO 2011, LNCS, vol. 6841, pp. 487–504. Springer (2011).
5. Coron, J.S., Naccache, D., Tibouchi, M: Public key compression and modulus switching for fully homomorphic encryption over the integers. EUROCRYPT'12. pp. 446–464. Springer (2012).
6. Cheon, J. H., Coron, J.-S., Kim, J., Lee, M. S., Lepoint, T., Tibouchi, M., and Yun A.: Batch Fully Homomorphic Encryption over the Integers, Advances in Cryptology – EUROCRYPT 2013, Springer, LNCS, Volume 7881, pp. 315-335. (2013).
7. Smart, N.P., Vercauteren, F: Fully homomorphic encryption with relatively small key and ciphertext sizes. PKC 2010, LNCS, vol. 6056, pp. 420–443. Springer (2010).
8. Brakerski, Z., Vaikuntanathan, V.: Efficient Fully Homomorphic Encryption from (Standard) LWE. FOCS (2011).
9. Brakerski, Z., Vaikuntanathan, V.: Fully Homomorphic Encryption for Ring-LWE and Security for Key Dependent Messages. CRYPTO 2011, LNCS, vol. 6841, pp. 505–524. Springer (2011).
10. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully Homomorphic Encryption without Bootstrapping. Cryptology ePrint Archive, Report 2011/277 (2011).
11. Gentry, C., Halevi, S.: Implementing Gentry's fully homomorphic encryption scheme. EUROCRYPT 2011, LNCS, vol. 6632, pp. 129–148. Springer (2011).
12. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic Evaluation of the AES circuit. Cryptology ePrint Archive, Report 2012/099 (2012).
13. Lauter, K., Naehrig, M., Vaikuntanathan, V.: Can Homomorphic Encryption be Practical? Cryptology ePrint Archive, Report 2011/405 (2011).
14. Moore, C., Hanley, N., McAllister, J., O'Neill, M., O'Sullivan, Elizabeth., and Cao, X.: Targeting FPGA DSP Slices for a Large Integer Multiplier for Integer Based FHE. Workshop on Applied Homomorphic Cryptography 2013. (2013).
15. Schönhage, A., Strassen, V.: Schnelle multiplikation grosser zahlen. Computing, vol. 7, no. 3, pp. 281–292. (1971).
16. Craven, S., Patterson, C., Athanas, P.: Super-sized multiplies: how do FPGAs fare in extended digit multipliers. 7th International Conference on Military and Aerospace Programmable Logic Devices. (2004).
17. Kalach, K., David, J.P.: Hardware implementation of large number multiplication by FFT with modular arithmetic, the 3rd International IEEE-NEWCAS Conference, pp.267 – 270. (2005).
18. Emmart, N., Weems, C.: High precision integer multiplication with a gpu using strassen's algorithm with multiple fft sizes. Parallel Processing Letters, vol. 21, no. 3, p. 359. (2011).
19. http://gmplib.org/manual/Multiplication-Algorithms.html#Multiplication-Algorithms.
20. Wang, W., Hu, Y., Chen, L., Huang, X., Sunar, B.: Accelerating fully homomorphic encryption using GPU. High Performance Extreme Computing (HPEC), pp. 1-5. IEEE (2012).
21. Cousins, D.B., Rohloff, K., Peikert, C., Schantz, R.: SIPHER: Scalable implementation of primitives for homomorphic encryption–FPGA implementation using Simulink. High Performance Extreme Computing Conference (2011).
22. Cousins, D.B., Rohloff, K., Peikert, C., Schantz, R.: SIPHER: An update on SIPHER (Scalable Implementation of Primitives for Homomorphic EncRyption) — FPGA implementation using Simulink. High Performance Extreme Computing Conference (2012).

23. Pöppelmann, T., Güneysu, T.: Towards Efficient Arithmetic for Lattice-Based Cryptography on Reconfigurable Hardware, LNCS, Volume 7533, pp. 139-158, Progress in Cryptology – LATINCRYPT 2012.
24. Cao, X., Moore, C., O'Neill, M., O'Sullivan, E., and Hanley, N.: Accelerating Fully Homomorphic Encryption over the Integers with Super-size Hardware Multiplier and Modular Reduction. Available from: http://eprint.iacr.org/2013/616.
25. Xilinx Product Specification: LogiCORE IP Multiplier v11.2. http://www.xilinx.com/support/documentation/ip_documentation/mult_gen_ds255.pdf.
26. Cheng, C., Parhi, K.K.: High-Throughput VLSI Architecture for FFT Computation, IEEE transactions on circuits and systems—ii: express briefs, vol. 54, No. 10. (2007).
27. Solinas, J. A.: Generalized Mersenne Numbers, Issue 39 of Research report (University of Waterloo. Faculty of Mathematics). (1999).
28. Barrett, P.: Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Advances in CRYPTO'86. pp. 311–323. Springer, (1987).