A Million-bit Multiplier Architecture for Fully Homomorphic Encryption

Yarkın Doröz

Worcester Polytechnic Institute,100 Institute Road, Worcester, MA 01609 USA

Erdinç Öztürk*

Istanbul Commerce University, Kucukyali, Istanbul, 34840, Turkey

Berk Sunar*

Worcester Polytechnic Institute,100 Institute Road, Worcester, MA 01609 USA

Abstract

In this work we present a full and complete evaluation of a very large multiplication scheme in custom hardware. We designed a novel architecture to realize a million-bit multiplication scheme based on the Schönhage-Strassen Algorithm. We constructed our scheme using Number Theoretical Transform (NTT). The construction makes use of an innovative cache architecture along with processing elements customized to match the computation and access patterns of the NTT–based recursive multiplication algorithm. We realized our architecture with Verilog and using a 90nm TSMC library, we could get a maximum clock frequency of 666 MHz. With this frequency, our architecture is able to compute the product of two million–bit integers in 7.74 milliseconds. Our data shows that the performance of our design matches

*Corresponding author

Email addresses: ydoroz@wpi.edu (Yarkın Doröz), eozturk@ticaret.edu.tr (Erdinç Öztürk), sunar@wpi.edu (Berk Sunar)

Preprint submitted to Elsevier *May 16, 2014*

that of previously reported software implementations on a high-end 3 GHz Intel Xeon processor, while requiring only a tiny fraction of the area.[1]
Keywords: Fully Homomorphic Encryption, Very–Large Number Multiplication, Number Theoretic Transform

---

1. Introduction

Arbitrary precision arithmetic has been studied for a long time, and it is a well-established research field. Since multiplication is the most compute–intensive basic operation, numerous algorithms for fast large–number multiplication has been introduced. Until 1960, school–book multiplication method was thought to be the fastest algorithm for multiplying two large numbers. In 1960, Karatsuba Algorithm was discovered and it was published in 1962 [16]. Although Karatsuba algorithm is much more efficient than school-book multiplication for large numbers, it is possible to optimize multiplication algorithm for very–large numbers even further. In 1971, Schönhage-Strassen Algorithm [10], which is an FFT-based large integer multiplication algorithm, was introduced. FFT-based algorithm is using floating-point arithmetic and this leads to rounding errors. For precise multiplication, we use Number Theoretic Transforms (NTT) instead of FFT.

Very–large integer arithmetic is used by a few number of applications. However, this does not change the fact that efficient implementations of very–large multiplication algorithms is an important open problem. For some important and popular research fields, such as primality testing and homomorphic encryption, the runtime and throughput of multiplication of

[1] An earlier short conference version of this paper appeared in [41].

two very large numbers is crucial for the practicality and the efficiency of the scheme.

As stated above, very–large integer multiplication is important for the field of Fully Homomorphic Encryption (FHE). The first proposed FHE scheme, which permits an untrusted party to evaluate arbitrary logic operations directly on the ciphertext, was introduced by Gentry and Halevi in [4]. This was an exciting development for the field of cryptographic research and led to tremendous amount of studies searching for further efficient FHE schemes. FHE holds great promise for numerous applications including private information retrieval and private search and data aggregation, electronic voting and biometrics. FHE's role is further amplified in the *cloud* where a server is shared by multiple users.

Although FHE schemes hold great promise, widespread applications of FHE will only be possible when efficient FHE implementations are available. Unfortunately, current FHE schemes, especially the one introduced by Gentry and Halevi in [4], have significant performance issues. None of the existing FHE proposals are considered to be practical yet, due to the extreme computational resource requirements. For instance, the Gentry and Halevi FHE [5], is reporting a $\sim$30 second latency for the re-encryption primitive on an Intel Xeon enabled server. Their scheme requires multiplication of operands that are a few million bits and performances of encryption, decryption and re-encryption evaluations are primarily determined by this operation. Existing software packages can handle integer operations at this magnitude (e.g. GNU Multiple Precision Arithmetic Library [7]). However, the prohibitive high cost of the very–large multiplier motivates the study of custom designed

architectures. This is the goal of this work and we are not aware of a custom designed architecture implementation for very–large integer multiplication.

While more efficient schemes are appearing in the literature, we would like to take a snapshot of the attainable hardware performance of the Gentry-Halevi FHE variant, by implementing a cost-efficient very–large integer multiplier using the FFT–based multiplication algoritm. We are motivated by the fact that the first commercial implementations of most public–key cryptographic schemes, e.g. RSA, Elliptic Curve DSA schemes have been via limited functionality hardware products due to the efficiency shortcomings on general purpose computers. We speculate that a similar growth pattern will emerge in the maturation process of FHEs. While the obvious efficiency shortcomings of FHE's are being worked out, we would like to pose the following question: *How far are FHE schemes from being offered as a hardware product?* Clearly answering this question would be a major overtaking deserving the evaluation of all FHE variants with numerous implementation techniques. Here, by providing strong area and performance numbers for the multiplier that we designed, we provide initial results for the FHE scheme of Gentry and Halevi.

In this work we present an implementation of our architecture realizing multiplication of two million-bit integers. We used a variation of the FTT–based Schönhage-Strassen Algorithm using NTT approach. Given the massive amount of data that needs to be efficiently processed, we identify data input/output and associate routing as one of the challenges in our effort. A secondary challenge is the recursive nature of NTT computations, which makes it hard to parallelize the architecture without impeding data

flow. We achieve a high–speed custom hardware architecture with a reasonable footprint that may be easily extended to realize the primitives of Gentry–Halevi FHE Scheme.

The rest of the paper is organized as follows. We present a brief review of the previous work in Section 2 and introduce the basic background in Section 3. We describe our approach in Section 4 and present the details of our design in Section 5. After the performance analysis in Section 6, we share the implementation results in Section 7.

2. Previous Work

Until recently, large integer multiplication implementations were mostly limited to few thousand–bits for classical encryption schemes. The introduction of Fully Homomorphic Encryption (FHE) schemes, especially Gentry and Halevi's FHE scheme, changed the focus to very–large integer multiplication algorithms. More groups started working on multiplication designs for different platforms with different optimization interests. In this section, we will give background information on previous implementations of traditional Large integer multiplication algorithms in 2.1. Also, in 2.2, we are giving background information on previous implementations of FHE schemes.

2.1. Traditional Large Integer Multipliers

For cryptographic applications, key sizes differ for different algorithms and schemes, to provide the same amount of security. For mainstream cryptographic applications, largest key sizes are used for public-key cryptography, such as RSA algorithm. These keys are currently in the range of 1024–bit

to 2048–bits, for reasonable security. However, FHE schemes are quite unusual requiring support for extremely long integer operands, from hundreds of thousands to millions of bits.

As we are discussing very–large integer arithmetic, which is not a well-studied research field, the natural place to turn for high performance *large* integer multiplication algorithms is the field of public-key cryptography. If we examine the efficiency and the methods of the algorithms being used for large–integer multiplication, we could extrapolate our findings to very–large integer multiplication schemes. Public key schemes such as RSA and Diffie-Hellman commonly use efficient multiplication algorithms that can handle operands of thousands of bits. Numerous efficient implementations supporting RSA is available in the literature[1] [2] [3].The thousand-bit multiplication algorithms are generally realized using two methods: classical multiplication and Karatsuba multiplication. For software implementations, due to constant-time requirements of the cryptographic schemes and the long and inefficient carry chains on the CPU pipeline, classical multiplication algorithm gives faster results. However, for hardware implementations, Karatsuba Algorithm is widely used. These implementations typically break down the long operands using a few iterations of the Karatsuba-Ofman algorithm [16]. The multiplication complexity of the Karatsuba-Ofman algorithm is sub-quadratic, i.e. $O(n^{\log_2 3})$ where n denotes the length of the operands. While presenting great savings, even the Karatsuba-Ofman algorithm becomes insufficient when operand sizes grow beyond tens of thousands of bits.

Although there is a significant amount of literature on FFT–based multiplication algorithms, we encountered very few that gave results for a detailed

and full implementation in hardware. One of the very few implementation examples is the architecture presented in [9]. They implement an FFT–based multiplication scheme with scalable inputs. This architecture consists only of one stage of the butterfly unit, which is not a complete design, and one FFT unit that achieves a 1024-bit FFT computation in 448 clock cycles. In the follow–up paper [12] the authors present an area–cost comparison of several multiplier architectures. Their implementation realizes a modular multiplication, using an architecture built around $\frac{3}{2}\log S$ butterfly operators and one accumulator in S clock cycles, where S is the degree of the polynomial that is used for FFT multiplication. This polynomial is the result of FFT–conversion of the integer to be multiplied.

In [13], the authors implement large–squaring operation utilizing a more complicated Number Theoretical Transform (NTT) based architecture. The authors claim that their squaring architecture can also be used for multiplication with minor modifications. They take the square of a 12–million digit number in 34 milliseconds using a XC2VP100 with a 80 MHz clock frequency. Their result is 1.76 times faster than a software implementation on Intel Pentium 4. However, since their design costs ten times more than a Pentium processor, the authors note that the cost of their design outweighs the performance gain.

Another design focusing on hardware implementation of very–large multiplication was proposed in [14]. This algorithm is more similar to the algorithms that we used in our implementation. For number of digits ranging from 2^5 to 2^{13}, their implementation is 25.7 times faster than software–based implementations, with an area cost of $9.05mm^2$. When the number of digits

is increased to 2^{21} (with a digit size of 4 bits) their implementation is 35.7 times faster than software–based implementations, with an area of $16.1mm^2$. Although this architecture achieves fast multiplication, the computations are performed over floating–point numbers, which introduces the possibility of an error. Indeed, the authors report error rates steadily increasing with increasing hamming-weight. They report error rates of about 0.025 and 0.06 for the operand lengths $4,000$ and $8,000$ bits, respectively. Clearly, the error rates will reach unacceptable levels for million–bit operands. Furthermore, for cryptographic applications, even a single bit error will have disastrous effects on the overall scheme.

Another very–large number multiplication implementation on FPGA was presented by Wang and Huang [28]. The authors propose an architecture for a 768K-bit FFT multiplier using a 64K-point finite field FFT as the key component. This component was implemented on both a Stratix V FPGA and an NVIDIA Tesla C2050 GPU and the FPGA implementation achieved a speed that is twice the speed of the implementation on the GPU [23].

2.2. GPU and FPGA Implementations of Homomorphic Encryption Schemes

There are currently several research groups working towards development of optimized FHE architectures targeting platforms alternative to software implementations, to improve the efficiency of Somewhat Homomorphic Encryption (SHE) and FHE schemes. Recent developments in this area show that efficient hardware implementations will drastically increase the efficiency of fully homomorphic encryption schemes. The performance of very–large multiplication, which is an important primitive of some FHE schemes, could be significantly improved through the use of FPGA technology. Compared

to ASIC design, FPGA technology offers flexibility at low cost. In addition, modern FPGAs include embedded hardware blocks, which are optimized for multiply-accumulate (MAC) operations, and thus can be exploited when implementing very–large multiplications.

Another alternative to software implementation that can improve the performance of SHE schemes is GPU implementations. An efficient GPU implementation of an FHE scheme was presented by Wang *et al.* [23]. The authors implemented the small parameter size version of Gentry and Halevi's lattice-based FHE scheme [18] on an NVIDIA C2050 GPU using the FFT algorithm, achieving speed–up factors of 7.68, 7.4 and 6.59 for encryption, decryption and the recryption operations, respectively. The FFT algorithm was used to target the bottleneck of this lattice-based scheme, namely the modular multiplication of very–large numbers, and the authors took advantage of the highly parallelized GPU architecture to accelerate the performance of the FHE scheme. However, the authors state that even with this speed–up, the implemented FHE scheme remains unpractical. In [42], the authors present an optimized version of their previos implementation. This modified method, implemented on an NVIDIA GTX 690, achieves speed–up factors of 174, 7.6 and 13.5 for encryption, decryption and the recryption operations, respectively, when compared to results of the implementation of Gentry and Halevi's FHE scheme [18] that runs on an Intel Core i7 3770K machine.

Other work has also focused on the use of efficient large integer multiplication implementations for many FHE schemes [18, 39, 40], in order to achieve faster hardware implementations. The use of a Comba multiplier[43], which can be implemented efficiently using the DSP slices of an FPGA [44], was

proposed by Moore *et al.* [26] to improve the performance of integer based FHE schemes [38] [40]. An FPGA implementation of a RLWE SHE scheme was also targeted by Cousins *et al.* [24] [25], in which Matlab Simulink is used to design the FHE primitives.

Lastly, Cao *et al.* [27] proposed a large-integer multiplier using integer-FFT multipliers combined with Barrett reduction to target the multiplication and modular reduction bottlenecks featured in many FHE schemes. The encryption step in the proposed integer based FHE schemes by Coron *et al.* [39, 40] were designed and implemented on a Xilinx Virtex-7 FPGA. The synthesis results show speed up factors of over 40 are achieved compared to existing software implementations of this encryption step [27]. This speed up illustrates that further research into hardware implementations could greatly improve the performance of these FHE schemes. An overview of the above mentioned multipliers for different platforms is given in Table 4.

3. Background

Common multiplication schemes (Karatsuba Algorithm, Classical schoolbook multiplication method) become infeasible for very-large number multiplications. For very-large operand sizes, efficient NTT-based multiplication schemes have been developed. SchönhageStrassen algorithm is currently asymptotically the fastest algorithm for very-large numbers. It has been shown that it outperforms classic schemes for operand sizes larger than 2^{17} bits [11].

3.1. Schönhage Strassen Algorithm

The Schönhage-Strassen Algorithm is an NTT-based large integer multiplication algorithm, with a runtime of $O(N \log N \log \log N)$ [10]. For an N-digit number, NTT is computed using the ring $R_N = \mathbb{Z}/(2^N + 1)\mathbb{Z}$, where N is a power of 2.

In [15], the algorithm is explained as follows:

Algorithm 1: Schönhage–Strassen Algorithm

Input: Multiplicand operands A and B, base $R = 2^t$

Output: Product C

1 Compute the NTT of the digits (with respect to the base) of A and B

2 Multiply the NTT results, component by component:

3   *Set* $C[i] = \mathsf{NTT}(A)[i] * \mathsf{NTT}(B)[i]$

4 Compute Inverse NTT of C

5   *Set* $C' = \mathsf{INTT}(C)$

6 Accumulate the carries

7   **if** $C'[i] \geq R$ **then**

8   *Set* $C[i+1] = C'[i+1] + \lfloor C'[i]/R \rfloor$

9   *Set* $C[i] = C'[i] \pmod{R}$

10 **return** C

In the algorithm, for the NTT evaluation we sample the numbers A and B into N-digits with a digit size of ϵ. By this sampling, we represent numbers A and B with tuples $(a_0, a_1, ..., a_{N-1})$ and $(b_0, b_1, ..., b_{N-1})$, respectively, where

each a_i is ϵ bits. NTT evaluation on these numbers is done as follows:

$$A_k^{NTT} = \sum_{i=0}^{N-1} w^{ik} a_i \quad \text{and} \quad B_k^{NTT} = \sum_{i=0}^{N-1} w^{ik} b_i \ .$$

where A^{NTT} and B^{NTT} are NTT-mapped tuples of A and B, respectively. The multiplications are modular multiplications with a selected prime p and w is a primitive root of p, i.e. $w^p = 1 \pmod{p}$. Here, the prime p has to be selected large enough to prevent overflow errors during the next phase of the NTT–based multiplication algorithm. For integer multiplication, the tuples in the NTT–form are multiplied to form another tuple C^{NTT} with elements C_k^{NTT}:

$$C_k^{NTT} = A_k^{NTT} \cdot B_k^{NTT} \pmod{p} \ .$$

Using the inverse-NTT we compute

$$C_k = \sum_{k=0}^{N-1} w^{-k} C_k^{NTT} \ .$$

As the last step, we can accumulate the carry additions to finalize the evaluation of C.

To realize the Schönhage-Strassen Algorithm efficiently, it is crucial to employ *fast* NTT and inverse-NTT computation techniques. We adopted the most common method for computing Fast Fourier Transforms (FFTs), i.e. the Cooley-Tukey FFT Algorithm [8]. The algorithm computes the Fourier Transform of a sequence X:

$$X_k = \sum_{j=0}^{N-1} x_j e^{-i2\pi k \frac{j}{N}} \ ,$$

by turning the length N transform computation into two $\frac{N}{2}$ size Fourier

Transform computations as follows

$$X_k = \sum_{\substack{m=0 \\ m \text{ even}}}^{N/2-1} x_{2m} e^{-i2\pi k \frac{m}{N/2}} + e^{\frac{-2\pi ik}{N}} \sum_{\substack{m=0 \\ m \text{ odd}}}^{N/2-1} x_{2m+1} e^{-i2\pi k \frac{m}{N/2}} \; .$$

We change $e^{\frac{-2\pi ik}{N}}$ with powers of w and perform the divisions into two halves with *odd* and *even* indices, recursively. With the use of fast transform technique, we can evaluate the Schönhage-Strassen Multiplication Algorithm in $O(N \log N \log\log N)$ time.

4. Our Approach

Our literature review has revealed that existing multiplier schemes are lacking some of the essential features we need for FHE schemes. A few references propose FFT based implementations of large integer multiplication techniques. These techniques exploit the convolution theorem to gain significant speedup over other multiplication techniques. To overcome the inefficiencies experienced in [13] and the noise problem experienced in [14] we chose to develop an application specific custom architecture based on the NTT transform.

The Schönhage-Strassen algorithm is the fastest and most efficient option for large integer multiplications. The algorithm has an asymptotic complexity of $O(N \log N \log\log N)$. Given the size of our operands the Schönhage-Strassen algorithm is perfectly suited to fulfill our performance needs. Another advantage of the algorithm is that it lends itself to a high degree of parallelization. A part of high level illustration of the parallel realization of the algorithm for short operands is shown in Figure 1. In the figure, shown

registers are the same registers which are shown separately for ease of view. We can add more reconstruction units in parallel with factor of the digit size.

In our implementation of the Schönhage-Strassen algorithm, the number of digits is $3 \times 2^{15} = 98304$. With the Cooley-Tukey approach, we form our butterfly circuit down to the size of $3 \times 2^2 = 12$ digits. This approach provides ample opportunities for parallelization. However, there are limits on how far we go exploiting the parallelism in a hardware realization. There are two reasons of this:

- The first reason for this is, the stage reconstruction computations are simple operations that take a few clock cycles, whereas we have huge data size to process. To overcome this obstacle we need an architecture that can handle a higher data bandwidth with rapid data access to perform calculations in parallel. Otherwise just by increasing the computational blocks while keeping restricted (low) bandwidth we will not be able to improve the performance.

- As for the second reason, in the algorithm the index range of the dependent digits double in each stage which requires significantly more routing on the digits. As the number of computational blocks and the stage number increases, too many collisions and overlaps occur in the routing for the VLSI design tools to handle.

Having a well designed high capacity cache is the key to achieving high performance large integer multiplier. Furthermore, the cache must be tailored to match of the computational elements. In our design we chose to incorporate $m = 4$ computation units, so that we can have some performance

boost and also avoid the routing problems that might occur. Even with $m = 4$ the bus width reaches 512-bits. In order to supply the computation blocks with sufficient data, we have chosen the cache size of the architecture as $N = 3 \times 2^{15} = 98304$. Moreover, to enable parallel reads without impeding the bandwidth, we divided the cache into $2 \times m$ sub-caches. It is possible to incorporate more computation units, but it will also make the control logic harder to design and create additional routing complications.

5. Design Details

5.1. Parameter Selection

The parameters for the implementation are based on the parameters in [15]. We choose 64-bit word size for fast and efficient computations, sampling size as $b = 24$ and the modulus as $p = 2^{64} - 2^{32} + 1$. The reason for the particular choice of p is that it allows us to realize a modular reduction using only a few primitive arithmetic operations. A 128-bit number is denoted as $z = 2^{96}a + 2^{64}b + 2^{32}c + d$. Using the selected p, we perform $z \pmod p$ operation as $2^{32}(b + c) - a - b + d$.

For the parameter N, we need to satisfy $\frac{N}{2}(2^b - 1)^2 < p$ to prevent the digit-wise overflows in the intermediary computations in NTT. Also, N should be big enough to cover million-bit multiplication with smallest possible value, so that the computation time is small. The best candidate for N is determined as $N = 3 \cdot 2^{15} = 98304$. As we use 24 bits for digit length, $3 \cdot 2^{15}$ digits hold a $1,179,648$–bit number for NTT–based multiplication (Although $3 \cdot 2^{15} \cdot 24$ is double this size, the number to be multiplied has to be truncated with 0s to double its size, therefor we can only multiply 2 $1,179,648$–bit

numbers by using $3 \cdot 2^{15}$ digits of 24 bits). This is the closest we could get to 1 million bits without losing efficiency for the NTT transform. Therefore, this parameter achieves one million-bit multiplication with minimum memory overhead.

Finally, we determine the Nth primitive root of unity, i.e. w. From the equation $w^N \equiv 1 \bmod p$, we determine $w = 3511764839390700819$.

In Cooley-Tukey FFT Algorithm, each recursive halving operation is referred as a stage and it is denoted as S_i, where i is the stage index. The size of the smallest NTT block is selected to be 12 entries and it is referred as the 0th stage, i.e. S_0. The remaining stages are reconstruction stages and require different arithmetic operations from the ones in S_0. As the nature of the NTT algorithm, in each reconstruction stage processing block size is doubled. Therefore, it takes 13 reconstruction stages to complete the NTT operation.

In terms of INTT operations, every stage and operation is identical to NTT. Only difference is selection of w'. It is computed as: $w' = w^{-1} \bmod p$.

5.2. Architecture Overview

Our architecture is composed of a data cache, a central control unit, two routing units and an function unit. The multiplication architecture is illustrated in Figure 2. The architecture is designed to perform a restricted set of special functions. There are four functions for handling the input/output transactions and three functions for arithmetic operations:

Sequential Load. It is used to store a million-bit number to the cache. The number is given in digits starting from the least significant digit and stored in sequence starting from the beginning of the cache.

Sequential Unload. The cache releases its contents starting from the least significant to most significant digit.

Butterfly Load. In NTT an important step is the distribution of the digits into the right indices using the butterfly operation. Starting from the least significant digit, the digits are inserted to the cache locations identified by the butterfly indices.

Scale & Unload. In the last step of the NTT–based multiplication computation, as a part of the INTT operation, the digits need to be scaled by $N^{-1} \pmod{p}$, as a part of the multiplication algorithm. After this operation, the carries are accumulated and the digits are scaled back to 24 bits. The Scale Unit overlaps scale and carry accumulation of the digits with output, since it is the final step in million–bit multiplication.

12x12 NTT/INTT. The smallest NTT/INTT computation is for 12 digits. 12x12 NTT/INTT Unit takes the digits sequentially and computes the 12 digit NTT/INTT by using simple shifts and addition operations.

Stage-Reconstruction. This function is used for reconstruction of the stages. According to the given input, it reconstructs an NTT or INTT stages. Reconstructions only complete a stage at a time, with the given stage index input. In order to complete a full reconstruction, i.e. completing evaluation of NTT, it is recalled for all 13 stages.

Inner-Multiplication. This operation is used to compute the digit-wise modular multiplications. For this we utilize the multipliers used in Stage-Reconstruction Unit.

Using the functions outlined above, we can compute the product of million-

bit numbers A and B using the following sequence of operations:

1. A is loaded into cache by using BUTTERFLY LOAD.
2. The NTT of number A, i.e. $\mathsf{NTT}(A)$, is computed by calling; first 12x12 NTT function, and afterwards STAGE-RECONSTRUCTION function for all stages.
3. $\mathsf{NTT}(A)$ is stored back to the RAM using SEQUENTIAL UNLOAD.
4. Using the same sequence of functions as above, we also compute $\mathsf{NTT}(B)$.
5. The cache can only hold the half of the digits of $\mathsf{NTT}(A)$ and $\mathsf{NTT}(B)$. Therefore, the numbers are divided into lower and upper halves:

$$\mathsf{NTT}(A) = \{\mathsf{NTT}(A)_h, \mathsf{NTT}(A)_l\}$$

$$\mathsf{NTT}(B) = \{\mathsf{NTT}(B)_h, \mathsf{NTT}(B)_l\}$$

6. We use SEQUENTIAL LOAD function to store $\mathsf{NTT}(A)_h$ and $\mathsf{NTT}(B)_h$.
7. Later on, an INNER MULTIPLICATION function is used to calculate the digit-wise modular multiplications:

$$C_h[i] = \mathsf{NTT}(A)_h[i] * \mathsf{NTT}(B)_h[i]$$

8. The result is stored back to the RAM by SEQUENTIAL UNLOAD.
9. We repeat above three steps to compute the lower part:

$$C_l[i] = \mathsf{NTT}(A)_l[i] * \mathsf{NTT}(B)_l[i]$$

10. The result digits, i.e. $C[i]$, are loaded into the cache by SEQUENTIAL LOAD. At this point the cache will contain the multiplication result, but still in the NTT form.

11. The result is converted into integer form by using, 12x12 INTT function which is followed by a complete STAGE-RECONSTRUCTION functions:

$$C' = \mathsf{INTT}(C[i])$$

12. In the last step, the result is scaled and the carries are accumulated by SCALE & UNLOAD function to finalize computation of C:

$$C[i+1] = C'[i+1] + \lfloor C'[i]/p \rfloor$$

5.3. Cache System

The size of the cache is important for the timing of multiplications. In each stage reconstruction process of the NTT algorithm, we need to match the indices of *odd* and *even* digits. The index difference of the *odd* and *even* digits for a reconstruction stage is computed as: $S_{i,diff} = 12 \cdot 2^{i-1}$, where i is the index of reconstruction stages, i.e. $1 \leq i \leq 14$. Since later stages require digits from distant indices, an adequate sized cache should be chosen to reduce the number of input/output transactions between the cache and RAM.

Lets call N' as the chosen cache size. Then, we can divide the N digits into $2^t = N/N'$ blocks, i.e. $N = \{N_{2^t-1}, N_{2^t-2}, \ldots, N_0\}$. Once a block is given as input, we can compute the reconstruction stages until $N' < S_{i,diff}$ for the ith stage. Then, stating from the ith stage, N_j requires digits from N_{j+1} which j is block index. So, we need to divide N_j and N_{j+1} into halves and match the upper halves of N_j with N_{j+1}, and lower halves of N_j with N_{j+1}. This matching process adds $2N'$ clock cycles for each block. Then, the total input/output overhead is evaluated as $2N \cdot \log_2(N/N')$, where $\log_2(N/N')$ is

the number of the stages that requires digit matching from different blocks. In our implementation, we aim to optimize the speed by selecting N' as N.

Although a huge sized cache is important for our design, a straight cache implementation is not sufficient to support parallelism. The main arithmetic functions utilized in the multiplication process, such as 12×12 NTT/INTT [2], STAGE-RECONSTRUCTION and INNER MULTIPLICATION, are highly suitable for parallelization. In order to achieve parallelization, the cache should be able to sustain required bandwidth for multiple units. In order to sustain the bandwidth, we build up the cache by combining small, equal size caches or as we refer them sub-caches. Combining these sub-caches on a top level, we can select the cache to be used as a single-cache or a multi-cache system. In case of linear functions, such as SEQUENTIAL LOAD, BUTTERFLY LOAD, etc., the cache works as a single-cache with one input/output ports, where as for parallel functions, it works as a multi-cache system with multiple input/output ports. The number of sub-caches should be equal to $2 \times m$ (double the size of multipliers in STAGE-RECONSTRUCTION UNIT) to eliminate access read/write to the same sub-cache in the reconstruction processes. Each sub-cache has a size of $N/(2 \times m)$ and we denote them as; $\{sc_0, sc_1, \ldots, sc_{2m-1}\}$.

5.4. Routing Unit

The ROUTING UNITS play an important role to match the *odd* and *even* digits to the arithmetic units. As stated previously, the index difference of

[2]In the design we choose to implement one unit, because its benefit is not worth the area cost.

the digits is $(12 \cdot 2^{i-1})$. Therefore, in last $\log 2m$ reconstruction stage, *odd* and *even* digits fall into different sub-cache. The assignment of sub-caches to the arithmetic units for each stage reconstruction is shown in Table 1. In the Table, arithmetic units are referred as $arith_i$, which i is the index number.

Clearly, each arithmetic unit is assigned to two sub-caches in each stage. From S_1 to S_{10}, *odd* and *even* indices are matched from the same sub-cache. Once an evaluation in sc_{2i} is finished it continues to process sc_{2i+1}. In rest of the stages, *odd* and *even* values are present in different sub-caches, so they are accessed by selecting sc_{i1} and sc_{i2} interchangeably. These assignments between sub-caches and STAGE RECONSTRUCTION UNITS are done by the CENTRAL CONTROL UNIT.

5.5. Function Unit

The function unit can be divided into three parts, i.e. the SCALE UNIT, the 12x12 NTT/INTT UNIT and the STAGE RECONSTRUCTION UNIT.

SCALER UNIT: As mentioned earlier, after the 12x12 INTT operation, a final step needs to be performed to compute the product. This final operation can be shown as $d_i \times N^{-1} + c_i \pmod p = \{c_{i+1}, r_i\}$. For our parameter choice $N^{-1} \pmod p$ = 0xFFFF555455560001 – a constant number with a special form that can be computed by simple shift and add operations with a low number of arithmetic operations. The unit takes digits in each clock cycle starting from the least significant digit d_0. The carry c_0 is set to zero in startup. After each scaling operation, the digit is added with a carry from the previous operation and least $b = 24$ bits of the 64-bit result is taken as the result r_i. The remaining bits are used as carry c_{i+1} in the next iteration.

12x12 NTT/INTT UNIT: This unit can be used for either computing the first stage of NTT or for INTT according to the required operation. This first NTT/INTT stage is basically computed using the NTT formula:

$$X_j = \sum_{i=0}^{n} (w')^{ji} \times d_i \pmod{p} \ . \quad (1)$$

In the equation, j is the index of the digit to be computed, and d_i refers to the digit value with index i. The parameter $w' = w^{2^{13}} \pmod{p}$, since the coefficient is squared in each stage. The parameter n is the smallest NTT block value to be evaluated and in our architecture it is chosen to be $n = 12$. The computation can be performed in two ways. First, all the powers of $(w')^{ij}$ can be pre-computed into a lookup table and reused during the multiplications. However, this will be a costly operation since we need 121 multiplications (coefficients with $(w')^0$ are ignored). When the entire million-bit operand is considered, it will take $121 \times N/n = 991232$ multiplication operations. Having $m = 4$ multipliers will decrease the computation time by a factor of four. However, each multiplication takes two clock cycles and the total computation will take $495,616$ clock cycles.

The overhead of the first stage can be decreased significantly by exploiting the special structure of the constant coefficients. Indeed, we can realize the modular multiplications with simple shift and add operations followed by a modular reduction at the end. Furthermore, note that in NTT $w' =$ 0x10000 and in INTT $w' =$ 0xFFFFEFFFF00010001 and therefore only a few additions and shifts are needed. These simple operations can be squeezed into few clocks and pipelined to optimize throughput. Also coefficients repeat after a while, since $(w')^{12} \equiv w^{3\ 2^{(15)}} \equiv 1 \pmod{p}$. This eases the construction

of the NTT/INTT units, since we can reuse the same circuit in the process. Due to pipelining all operations can be performed in N clock cycles if we neglect the initial cycles needed to fill the pipeline to get the first output.

Stage Reconstruction Unit: This unit holds 64 bit multipliers and is responsible for the operation of two functions: Stage-Reconstruction and Inner Multiplication. The architecture is more complex than the other units in the Function Unit. It is composed of a control unit, and a coefficient block and an arithmetic unit. The architecture is illustrated in Figure 3.

In an Inner Multiplication function, two digits are read from the cache and fed to the $Coeff$ and odd input lines. Since we are reading digits from each sub-cache one at a time, we have two clock cycles to perform a 64-bit multiplication. In order to reach higher frequencies, we used two 32-bit multipliers and extended them to construct the 64-bit multipliers. Using a classical high–school approach, we are able to perform a 64-bit multiplication operation in 2 cycles. Since cache I/O speed is the bottleneck here and we cannot read faster than 2 cycles, utilizing faster multiplication algorithms, such as Karatsuba Multiplication algorithm, would not have improved the performance. The Inner Multiplication function computes a 64-bit multiplication, additions and a modular reduction. We pipelined the architecture to increase the clock frequency and the throughput. The unit can output a modular multiplication product in every two clock cycles after the initial startup cost of the pipeline. The whole function takes $\frac{N}{2}$ multiplications and with m multipliers it will cost $\frac{N}{m}$ clock cycles.

In the Stage-Reconstruction function, a more complex control mech-

anism is required compared to other functions. In each stage we need to compute

$$\begin{aligned} O_{i,j} &= E_{i-1,j} - O_{i-1,j} \times w_{i-1}^{j \pmod{n_{i-1}}} \pmod{p}, \\ E_{i,j} &= E_{i-1,j} + O_{i-1,j} \times w_{i-1}^{j \pmod{n_{i-1}}} \pmod{p} . \end{aligned} \tag{2}$$

In the equation, i denotes the stage index from 1 to 13, j denotes the index of the digits, w_i is the coefficient of stage $i-1$ and finally n_{i-1} is the modular reduction to select the appropriate power of the w_i. We may compute the terms n_i iteratively as $n_{i+1} = 2 \times n_i$ where $n_0 = 12$. Ideally we need to store all the coefficients along with the *odd* digits. However this will require another large cache of size $(12+24+48+\cdots+49152) \times N$ digits. The cache can be reduced to almost half in size, since during the stage computations each stage uses all of the coefficients from the previous stage. Therefore, the size can be reduced to $12 + (24 + 48 + \cdots + 49152)/2 = 49152 \times N$. In order to reduce the storage requirement, we can compute the coefficients using multiplications efficiently by using pre-stored coefficients as follows:

1. The coefficients required in two consecutive stages are as follows: $S_{i+1} : w_i^0, w_i^1, \ldots, w_i^{n_i}$ and $S_{i+2} : w_{i+1}^0, w_{i+1}^1, \ldots, w_{i+1}^{n_{i+1}}$.
2. Then $S_{i+2} :\ w_{i+1}^0, w_{i+1}^1, \ldots, w_{i+1}^{2n_i}$ since it holds that $n_{i+1} = 2 \times n_i$.
3. Further, $S_{i+2} :\ w_i^{\frac{0}{2}}, w_i^{\frac{1}{2}}, \ldots, w_i^{\frac{2n_i}{2}}$, since $w_i = w_{i+1}^2$.
4. This shows that half of the coefficients of S_{i+2} are same as S_{i+1} and the other half are the square roots of the coefficients of S_{i+1}.
5. We can compute the square roots by multiplying each w_i^j with w_{i+1}^1.

Using the properties described above, we construct the Coefficient Table by storing two columns of coefficients. In the first column, since our

smallest computation block is 12, we compute and store all w_0^j coefficients for $11 \geq j \geq 0$. We denote these coefficients as $w_{first,i}$, where i denotes the index of the coefficient. In the second column, for each of the remaining stages we compute and store w_i^1. The second column coefficients are denoted by $w_{second,i}$. This makes a total of 24 coefficients which we can use to compute any of the w_i^j values. When we include also the coefficients for the INTT operations, our table contains 48 coefficients. The computation of an arbitrary coefficient using the table can be achieved as

$$w_i^j = w_{first,l} \times \prod_{t=0}^{i} w_{second,t}^{e} \ .$$

The values of l and e are functions of i and j. Also e is a value equal to 1 or 0. Therefore we can omit the multiplications whenever $e = 0$. The total number of multiplications for computating $w_i^j \times O_i$ can be calculated as follows:

1. In every reconstruction stage we start by multiplying *odd* digits with $w_{first,l}$'s. This step makes a total of $\frac{N}{2}$ multiplications.
2. Apart from the first reconstruction stage, in each stage we also require coefficients from $w_{second,0}$ to $w_{second,i-1}$. Since we cannot store the coefficients, in each stage we need to rebuild the previous stage coefficients to build up the coefficients. We are using half of the previous stage values so in each stage we need $\frac{N}{4}$ additional multiplications.
3. The total multiplications will then becomes $\sum_{i=0}^{i=12}(\frac{N}{2}+i\times\frac{N}{4}) = 26\times N$.

The Stage Reconstruction architecture works with the following steps: The Central Control Unit sends an input bit sequence as {opcode, stage_level, digit_index} to the Stage Reconstruction Control Unit.

The opcode defines whether the function is an NTT, INTT or an Inner Multiplication operation. If it is a Stage-Reconstruction function, it checks the stage_level bits to determine the computing stage and digit_index bits to determine the starting point of the computation from the N digit range. These bits are necessary especially if there are multiple Stage Reconstruction Units. Each unit needs to determine its own startup coefficient, i.e. w_i^j, so they initialize different $w_{first,l}$ and $w_{second,t}$. Also another important initialization is the mul_only signal (for multiplication only), which is crucial for computing $w_i^j \times O_i$. In each multiplication of an *odd* digit with a coefficient means, either the computation of $w_i^j \times O_i$ is completed or it still needs more multiplications. According to the requirement, the mul_only signal is used for enabling $E_{i,j} \pm O_{i,j} \times w_i^{j \pmod{n_i}}$ calculation. If the coefficient does not need any more processing, the new *odd* and *even* values are stored in the cache. Otherwise, the *even* digit is not updated and the *odd* digit is updated using the rule $O'_{i+1,j} = O'_{i,j} \times w_{second,l}$ and stored into the cache. Here $O'_{i,j}$ denotes previously updated *odd* digit with a coefficient. In order to complete the computation of partially completed coefficients, the system is fed with *even* and O' digits, and with a new coefficient. This process continues until all the values are computed for a stage.

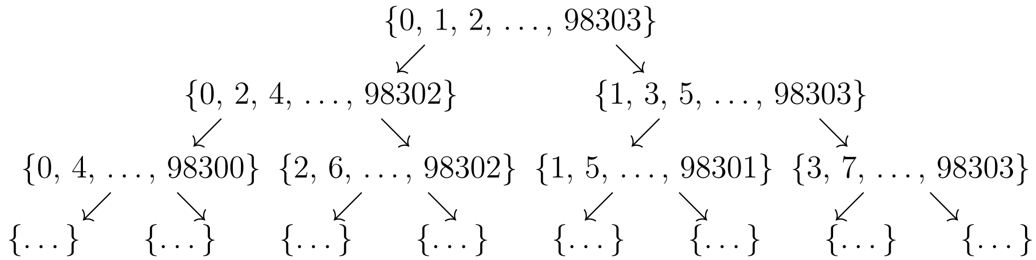
5.6. Central Control Unit

The Central Control Unit contains a state machine to handle the functions that are given as inputs. It is mainly responsible to start the requested function given as input to the system. Since 12×12 NTT/INTT Unit and Scale Unit consist only of datapath, they are only controlled with start and ready signals. As mentioned earlier, the Stage Recon-

struction Unit requires more complex operations, so each one is managed by their own control logic and each only requires a bit sequence {opcode, stage_level, digit_index} from the Central Control Unit. These bit sequences are pre-determined sequences according to the number of Stage Reconstruction Units and the cache size.

The Central Control Unit also handles input/output addressing of the sub-caches. Sequential functions require incremental addressing for each sub-cache. In Stage-Reconstruction function, the addressing is computed according to the stage level, which is basically updated with the index range of dependent *odd* and *even* digits.

The addressing for the Butterfly Load function requires a bit more complex approach compared to incremental addressing. For each level the butterfly operation divides given input block into even and odd indices recursively. For instance:



The N digits are divided into smaller blocks until the digit size reduces to 12. This operation creates 8192 blocks with 12 digits that forms a clear pattern that can be easily constructed. In our construction $N = 98304$ the pattern formed above can be constructed using lookup tables consisting of 35 elements.

6. Performance Analysis

The latency of each functional block in cycles is given in Table 2. Any LOAD or UNLOAD function takes N clock cycles, where N is the number of digits. Since the architecture consist of one 12×12 NTT/INTT function block the functions take N clock cycles per function. For the intermediate NTT/INTT operations, each stage has varying number of clock cycles to complete its operation. For the given stage with index i, i.e. S_i, the total number of clock cycles for that stage is $\lceil \frac{N}{m} \rceil + \lceil \frac{N}{2m} \rceil \times (i-1)$, where m denotes the total number of multipliers. The INNER MULTIPLICATION operation takes $\lceil \frac{N}{m} \rceil$ clock cycles.

In order to perform a complete multiplication, two complete NTT operations, two INNER MULTIPLICATION operations and one INTT operation need to be performed. A complete NTT or INTT operation can be performed as follows:

1. The number is loaded into the cache by using BUTTERFLY LOAD operation.
2. 12×12 NTT/INTT operation is performed.
3. STAGE RECONSTRUCTION operations are performed for stages 1 to 13.
4. A SCALE & UNLOAD is used to carry out the digit-wise additions, carry propagations and to scale the product.

The total number of clock cycles for a NTT/INTT operation will take N cycles per load and unload operations, N cycles for 12×12 NTT/INTT operation and each stage reconstruction will add $\lceil \frac{N}{m} \rceil + \lceil \frac{N}{2m} \rceil \times (i-1)$ cycles

simplified further as

$$\sum_{i=1}^{i=13}\left(\lceil\frac{N}{m}\rceil+\lceil\frac{N}{2m}\rceil\times(i-1)\right)=13\lceil\frac{N}{m}\rceil+78\lceil\frac{N}{2m}\rceil\ .$$

We need an additional $3N$ cycles to load one operand, for the 12×12 NTT computation and to unload the final result. The total number of cycles to compute the NTT of one operand becomes $\lceil\frac{52N}{m}\rceil+3N$. During the multiplication, we need to run NTT/INTT three times. An Inner Multiplication operation will take N cycles per load and unload operation and $\lceil\frac{N}{m}\rceil$ clock cycles for the intermediate multiplication operations. Note that since our cache can only hold one operand at a time we need to perform two Inner Multiplication operations. Hence, one complete large integer multiplication will take two NTT, two Inner Multiplication and one INTT operations resulting in a total of $158\lceil\frac{N}{m}\rceil+13N$. For instance, for a design with $m=4$ multiplier units we need in total $2\times 16N+2\times\frac{9N}{4}+16N=52.5N$ clock cycles. The cycle counts for the functional block and for large integer multiplication is given in Table 2.

6.1. Scalability of the Proposed Multiplier

Here we briefly study the the scalability of the proposed multiplier. We would like to be able to reach an optimal design point without unnecessarily crippling the time performance. In Table 3 we list the number of cycles and the cycle times number of multipliers units, in other words the time area product, for various choices of number of multiplier units incorporated in the large multiplier design. Clearly, increasing m decreases the time required to perform the computation. However, we see diminishing returns due to the constant term in the complexity equation, i.e. $158\lceil\frac{N}{m}\rceil+13N$.

To obtain the small improvement, we pay significantly in area, reducing overall efficiency as evidenced in the time area product shown in the last column. This suggests that a better optimization strategy is to simply instantiate the large NTT multipliers with a small number of multiplier units, e.g. $m = 4$ or $m = 8$, and then spend more area by replicating the large multipliers to gain linear improvement in the overall FHE primitive execution times for a linear investment in area.

7. Implementation Results

In our design we fixed the number of digits as $N = 98304$. The resulting architecture is capable of computing the product of two $1,179,648$-bit integers in 5.16 million clock cycles. We modeled the proposed architecture into Verilog modules and synthesized Synopsis Design Compiler using the TSMC 90 nm cell library. The architecture reaches at a maximum clock frequency of 666 MHz and can compute the product of two $1,179,648$-bit integers in 7.74 ms.

While it is impossible to make a fair comparison between an application specific architecture and other types of platforms, i.e. a general purpose microprocessor, GPUs and FPGAs, we find it useful to state the results side-by-side as shown in Table 4. The NVIDIA C2050 and NVIDIA GTX 690 GPUs are high-end expensive processors and consume significantly more area than our design; to be precise the GPUs contain ~ 18.7 and ~ 43.7 times more equivalent gates respectively. In another perspective, we are able to fit 18 or 43 of our multipliers into the same area that the GPUs occupy. Similarly, the Stratix V FPGA implementation [28] is an expensive high end

DSP oriented FPGA with significant on-chip memory. Compared to the proposed design a rough back of the envelope calculation reveals a 23.7 times larger footprint. The proposed design occupies only a tiny fraction of the footprint compared to the other proposals; in operations like recryption we can surpass at timing by implementing more of our multipliers and running them in parallel or they can be used for running multiple recryption processes. Although our multiplier is approximately 12 times slower, additional multipliers amortize the timing up to 3.5 times.

To gain more insight we focus more closely into the architecture. The main contributor to the footprint of the architecture was the 768 Kbyte cache which requires 26.5 million gates. In contrast, the footprint of the functional blocks is only 0.2 million gates. The total area of the design comes 26.7 million equivalent gates. With the time performance summarized in Table 5 our architecture matches the speed of Intel Xeon software [5] running the NTL software package [17] which features a very similar NTT-based multiplication algorithm.

8. Conclusion

We presented the first full evaluation of a million–bit multiplication scheme in custom hardware. For this, we introduced a novel architecture for efficiently realizing multi-million bit multiplications. Our design implements the Schönhage-Strassen Algorithm optimized using the Cooley-Tukey FTT technique to speed up the NTT conversions. When implemented in 90 nm technology, the architecture is capable of computing a million–bit multiplication in 7.74 milliseconds while consuming an area of only 26 million gates.

Our architecture presents a viable alternative to previous proposals, which either lack the required performance level or cannot provide error–free multiplication for the operand lengths we are seeking to support. Finally, we presented performance estimates for the Gentry-Halevi FHE primitives, assuming the large integer multiplier is used as a coprocessor. Our estimates show that the performance of our design matches the performance of previously reported software implementations on a high end Intel Xeon processor, while requiring only a tiny fraction of the area. Therefore, our evaluation shows that much higher levels of performance may be reached by implementing FHE in custom hardware thereby bringing FHE closer to deployment.

Acknowledgment

Funding for this research was in part provided by the US National Science Foundation CNS Award #1117590. Funding was also provided by The Scientific and Technological Research Council of Turkey, project number 113C019.

[1] C. K. Koc., High-Speed RSA Implementation, TR 201, RSA Laboratories, 73 pages, November 1994.

[2] C. K. Koc., RSA Hardware Implementation, TR 801, RSA Laboratories, 30 pages, April 1996.

[3] Shay Gueron, Efficient software implementations of modular exponentiation, Journal of Cryptographic Engineering Volume 2, Issue 1 , pp 31-43

[4] C. Gentry, A Fully Homomorphic Encryption Scheme, Ph.D. thesis, Department of Computer Science, Stanford University, 2009.

[5] Craig Gentry and Shai Halevi, Implementing Gentry's Fully-Homomorphic Encryption Scheme, In EUROCRYPT 2011, LNCS vol. 6632, pages 129-148, Springer 2011.

[6] D. Cousins, K. Rohloff, C. Peikert, and R. Schantz, "SIPHER: Scalable implementation of primitives for homomorphic encryption–FPGA implementation using Simulink," HPEC 2011.

[7] The GNU Multiple Precision Arithmetic Library, GMP 5.0.5, `http://gmplib.org/`.

[8] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. Mathematics of computation, 19(90):297-301, 1965.

[9] K. Kalach and J.P. David, Hardware implementation of large number multiplication by FFT with modular arithmetic, IEEE-NEWCAS Conference, 2005. The 3rd International, pp. 267–270, 19-22 June 2005.

[10] Arnold Schönhage and Volker Strassen, Schnelle Multiplikation grosser Zahlen, Computing 7(3), vol 7, Springer Wien, Sept. 1971, 281–292.

[11] Luis Carlos Coronado Garcìa, Can Schönhage multiplication speed up the RSA decryption or encryption?, MoraviaCrypt, 2007.

[12] J.P. David, K. Kalach, and N. Tittley, Hardware Complexity of Mod-

ular Multiplication and Exponentiation, IEEE Transactions on Computers, vol.56, no.10, pp.1308-1319, Oct. 2007.

[13] S. Craven, C. Patterson, and P. Athanas, Super-sized multiplies: how do FPGAs fare in extended digit multipliers? in Proceedings of the 7th International Conference on Military and Aerospace Programmable Logic Devices (MAPLD'04), Washington, DC, USA, September 2004.

[14] S. Yazaki and K. Abe, "An Optimum Design of FFT Multi-Digit Multiplier and Its VLSI Implementation," Bulletin of the University of Electro-Communications, Vol.18, No.1 and 2, pp.39-46, Jan. 2006.

[15] N. Emmart and C.C. Weems, High Precision Integer Multiplication with a GPU Using Strassen's Algorithm with Multiple FFT Sizes, presented at Parallel Processing Letters, pp. 359-375, 2011.

[16] A. Karatsuba and Yu. Ofman, Multiplication of Many-Digital Numbers by Automatic Computers, Proceedings of the USSR Academy of Sciences 145: 293-294, 1962.

[17] V. Shoup, NTL: A Library for doing Number Theory, Library Version 5.5.2, `http://www.shoup.net/ntl/`.

[18] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," in *EUROCRYPT*, 2011, pp. 129–148.

[19] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," *IACR Cryptology ePrint Archive*, vol. 2012, p. 99, 2012.

[20] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully homomorphic encryption without bootstrapping," *Electronic Colloquium on Computational Complexity (ECCC)*, vol. 18, p. 111, 2011.

[21] N. P. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," *IACR Cryptology ePrint Archive*, vol. 2011, p. 133, 2011.

[22] J.-S. Coron, T. Lepoint, and M. Tibouchi, "Batch fully homomorphic encryption over the integers," *IACR Cryptology ePrint Archive*, vol. 2013, p. 36, 2013.

[23] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption using GPU," in *HPEC*, 2012, pp. 1–5.

[24] D. Cousins, K. Rohloff, R. Schantz, and C. Peikert, "SIPHER: Scalable implementation of primitives for homomorphic encrytion," Internet Source, September 2011.

[25] D. Cousins, K. Rohloff, C. Peikert, and R. E. Schantz, "An update on SIPHER (scalable implementation of primitives for homomorphic encRyption) - FPGA implementation using simulink," in *HPEC*, 2012, pp. 1–5.

[26] C. Moore, N. Hanley, J. McAllister, M. O'Neill, E. O'Sullivan, and X. Cao, "Targeting FPGA DSP slices for a large integer multiplier for integer based FHE," *Workshop on Applied Homomorphic Cryptography*, vol. 7862, 2013.

[27] Xiaolin Cao, Ciara Moore, Mire O'Neill, Elizabeth O'Sullivan, and

Neil Hanley., Accelerating fully homomorphic encryption over the integers with super-size hardware multiplier and modular reduction, IACR Cryptology ePrint Archive (2013)

[28] W. Wang and X. Huang, “FPGA implementation of a large-number multiplier for fully homomorphic encryption,” in *ISCAS*, 2013, pp. 2589–2592.

[29] C. Gentry and S. Halevi, “Fully homomorphic encryption without squashing using depth-3 arithmetic circuits,” *IACR Cryptology ePrint Archive*, vol. 2011, p. 279, 2011.

[30] N. P. Smart and F. Vercauteren, “Fully homomorphic encryption with relatively small key and ciphertext sizes,” in *Public Key Cryptography*, 2010, pp. 420–443.

[31] Z. Brakerski and V. Vaikuntanathan, “Efficient fully homomorphic encryption from (standard) LWE,” in *FOCS*, 2011, pp. 97–106.

[32] C. Gentry, S. Halevi, and N. P. Smart, “Better bootstrapping in fully homomorphic encryption,” *IACR Cryptology ePrint Archive*, vol. 2011, p. 680, 2011.

[33] Craig Gentry and Shai Halevi and Nigel P. Smart, Fully homomorphic encryption with polylog overhead, IACR Cryptology ePrint Archive, vol. 2011, p. 566, 2011.

[34] O. Regev, “On lattices, learning with errors, random linear codes, and cryptography,” in *STOC*, 2005, pp. 84–93.

[35] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *IACR Cryptology ePrint Archive*, vol. 2012, p. 230, 2012.

[36] Z. Brakerski and V. Vaikuntanathan, "Fully homomorphic encryption from ring-LWE and security for key dependent messages," in *CRYPTO*, 2011, pp. 505–524.

[37] K. Lauter, M. Naehrig, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" *Cloud Computing Security Workshop*, pp. 113–124, 2011.

[38] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *EUROCRYPT*, 2010, pp. 24–43.

[39] J.-S. Coron, A. Mandal, D. Naccache, and M. Tibouchi, "Fully homomorphic encryption over the integers with shorter public keys," in *CRYPTO*, 2011, pp. 487–504.

[40] J.-S. Coron, D. Naccache, and M. Tibouchi, "Public key compression and modulus switching for fully homomorphic encryption over the integers," in *EUROCRYPT*, 2012, pp. 446–464.

[41] Y. Doröz, E. Öztürk, and B. Sunar, "Evaluating the hardware performance of a million-bit multiplier," in *Digital System Design (DSD), 2013 16th Euromicro Conference on*, 2013.

[42] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the

feasibility of fully homomorphic encryption," *IEEE Transactions on Computers*, vol. 99, no. PrePrints, p. 1, 2013.

[43] P. G. Comba, "Exponentiation cryptosystems on the IBM PC," *IBM Systems Journal*, vol. 29, no. 4, pp. 526–538, 1990.

[44] T. Güneysu, "Utilizing hard cores of modern FPGA devices for high-performance cryptography," *J. Cryptographic Engineering*, vol. 1, no. 1, pp. 37–55, 2011.

List of Figures

1 A part of fully parallel circuit realizing the Schönhage-Strassen algorithm 40
2 Overview of The Large Integer Multiplier 41
3 Stage Reconstruction Unit 42
4 ALU of The Stage Reconstruction Unit 43

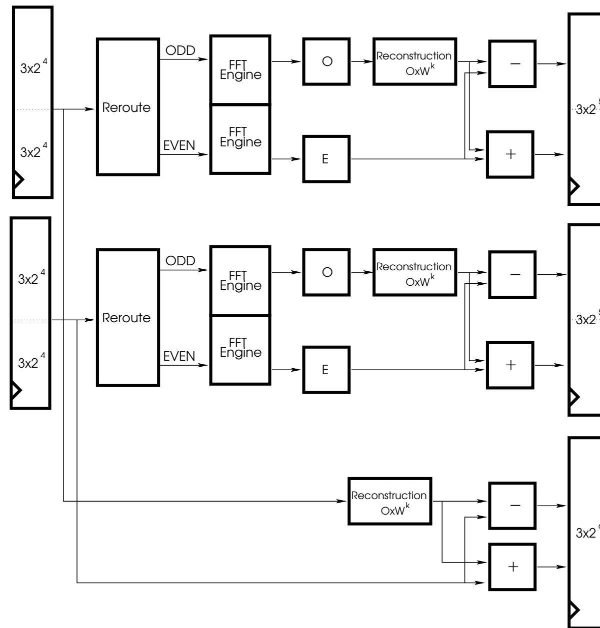


Figure 1: A part of fully parallel circuit realizing the Schönhage-Strassen algorithm

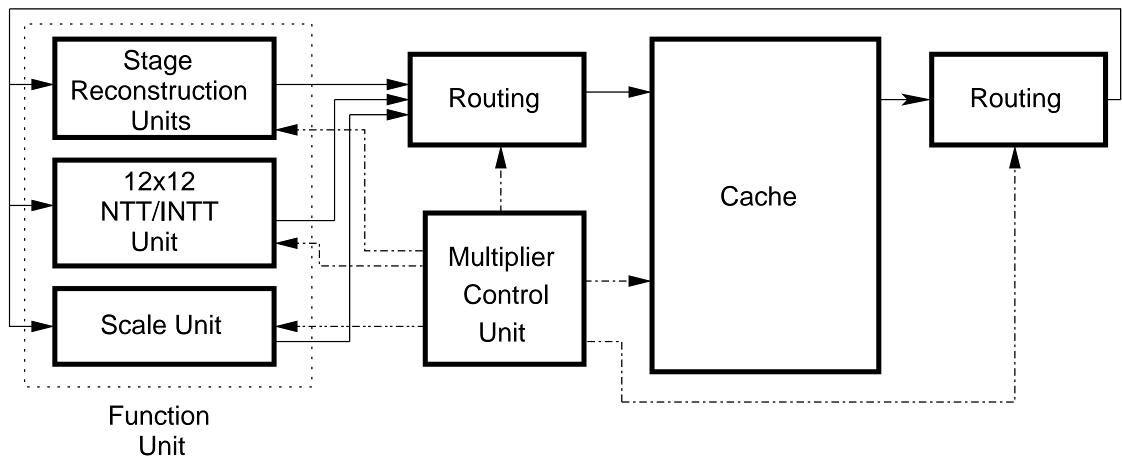


Figure 2: Overview of The Large Integer Multiplier

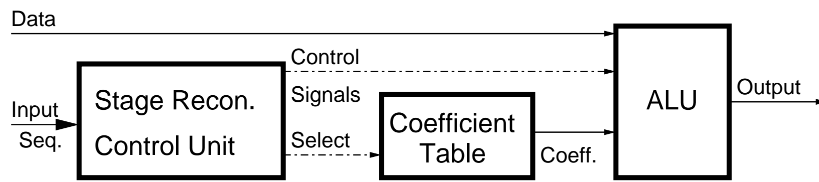


Figure 3: Stage Reconstruction Unit

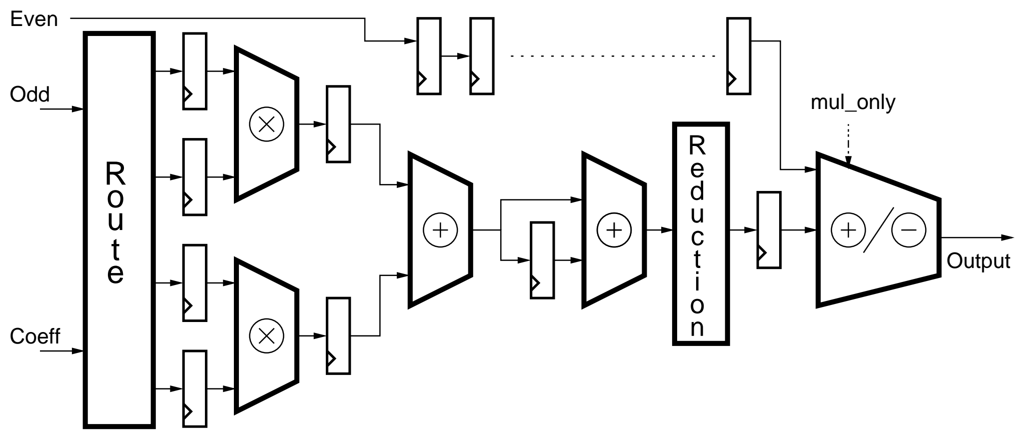


Figure 4: ALU of The Stage Reconstruction Unit

List of Tables

1 Assignment Table 45
2 Clock Cycle Counts of Functional Blocks 46
3 Clock Cycle Counts and Time Area Product for Various Number of Multiplier Units 47
4 Comparison of Timings for FFT Multipliers 48
5 FHE Performance Estimates for $m = 4$ multiplier units (M=mult., MM=modular mult.) 49

Table 1: Assignment Table

	$\mathbf{arith_0}$	$\mathbf{arith_1}$	$\mathbf{arith_2}$	$\mathbf{arith_3}$
S_1-S_{10}	sc_0-sc_1	sc_2-sc_3	sc_4-sc_5	sc_6-sc_7
S_{11}	sc_0-sc_1	sc_2-sc_3	sc_4-sc_5	sc_6-sc_7
S_{12}	sc_0-sc_2	sc_1-sc_3	sc_4-sc_6	sc_5-sc_7
S_{13}	sc_0-sc_4	sc_1-sc_5	sc_2-sc_6	sc_3-sc_7

Table 2: Clock Cycle Counts of Functional Blocks

NTT(A) , NTT(B)	2 Butterfly Load	$2N$
	2 12×12 NTT	$2N$
	2 Stage-Recon	$104\lceil \frac{N}{m} \rceil$
	2 Sequential Unload	$2N$
AxB	2 Sequential Load	$2N$
	2 Inner Multiplication	$2\lceil \frac{N}{m} \rceil$
	2 Sequential Unload	$2N$
INTT(AxB)	Butterfly Load	N
	12×12 INTT	N
	Stage-Recon	$52\lceil \frac{N}{m} \rceil$
	Scale Unload	N
TOTAL		$158\lceil \frac{N}{m} \rceil + 13N$

Table 3: Clock Cycle Counts and Time Area Product for Various Number of Multiplier Units

m	Total Cycles	Time Area ($m\times$ cycles)
4	$52.5N$	$210N$
8	$32.7N$	$262N$
16	$22.8N$	$366N$
32	$17.9N$	$574N$
64	$15.4N$	$990N$

Table 4: Comparison of Timings for FFT Multipliers

Design	**Platform**	**Mult.** (in ms)	**Area** (in Million Equiv. Gates)
Proposed FFT multiplier	90nm TSMC	7.74	26.7
FFT multiplier / reduction [23]	NVIDIA C2050 GPU	0.765	500
FFT multiplier / reduction [42]	NVIDIA GTX 690	0.583	1166
Only FFT Transform [28]	Stratix V FPGA	0.125	633

Table 5: FHE Performance Estimates for $m = 4$ multiplier units (M=mult., MM=modular mult.)

	Mul	**Barrett Red**	**Dec**	**Enc**	**Recrypt**
# of Ops	1 M	2 M	1 MM	90 MM	1354 MM
Ours	7.74 ms	15.4 ms	23.2 ms	2.09 s	31.4 s
GH [5]	6.67 ms	13.3 ms	20 ms	1.8 s	32 s