AREA-TIME PRODUCT EFFICIENT RNS POLYNOMIAL BASE EXTENSION ON FPGA

by
SELİM KIRBIYIK

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
July 2025

AREA-TIME PRODUCT EFFICIENT RNS POLYNOMIAL BASE EXTENSION ON FPGA

Approved by:

Prof. ERKAY SAVAŞ .....
(Thesis Supervisor)

Prof. ÖZCAN ÖZTÜRK .....

Assoc. Prof. SUJOY SINHA ROY .....

Date of Approval: July 22, 2025

ABSTRACT

AREA-TIME PRODUCT EFFICIENT RNS POLYNOMIAL BASE EXTENSION ON FPGA

SELİM KIRBIYIK

Computer Science and Engineering M.Sc. Thesis, July 2025

Thesis Supervisor: Prof. Erkay Savaş

Keywords: FPGA, FHE, RNS, BFV, accelerator

Homomorphic Encryption (HE) allows us to perform privacy-preserving processing of data by computing on ciphertexts. Using HE can eliminate the trust required to offload computation to third parties that have more scalable computational power. A barrier to adopting HE in more practical processing of data is its inherent computational complexity. Current HE schemes such as Brakerski-Fan-Vercauteren (BFV) can be accelerated through loosely coupled accelerator devices to accommodate the performance requirements of even more applications. Developing an accelerator for HE requires implementation of algorithms that have irregular memory accesses, such as Number Theoretic Transform (NTT). Even after accelerating the NTT algorithm, the communication overhead between the host and the device offsets some of the benefits of this acceleration. To achieve further performance gains, the implementation of a larger set of arithmetic operations is required. One such operation is base extension, needed when the Residue Number System (RNS) is utilized to perform efficient arithmetic of larger integers. For example, the modulus used for the ciphertext Q is chosen as 1747-bits to satisfy the 128-bit security level with ring dimension $N = 2^{16}$. Utilizing RNS, we can choose 28×64-bit primes or 55×32-bit primes to satisfy the required parameters. During homomorphic multiplication, our results may not fit within the range afforded by the current RNS base. Then, a base extension is required before the operation to increase the representation range. In a complete HE accelerator, the NTT unit produces residues that will be consumed by the base extension unit and vice versa to compute homomorphic multiplication operations on the accelerator device. By not communicating back to the host, the data movement overheads will be avoided. Our base extension implementation is

optimized for its Area-Time Product (ATP) metric for HE accelerators that have multiple units competing for device programmable logic area. We present a compile-time configurable hardware generator for exact base extension with parameters for the available memory bandwidth on the device. The design features a scalable architecture that decouples performance from the underlying base extension arithmetic. A compile-time tunable throughput parameter can increase the performance of the operation at the cost of additional logic area. We provide our Field Programmable Gate Array (FPGA) utilization results for ring sizes from 2^{12} to 2^{16}. We compare our base extension architecture to architectures available in the literature. We demonstrate comparisons with the state-of-the-art open source HE software library OpenFHE against our FPGA implementation and show that our implementation achieves a $\times 10 - 17$ speedup over the software implementation.

ÖZET

ALAN-ZAMAN FAKTÖRÜ AÇISINDAN VERİMLİ RNS POLİNOM TABAN GENİŞLETME FPGA DONANIMI

SELİM KIRBIYIK

Bilgisayar Bilimleri ve Mühendisliği Yüksek Lisans Tezi, Temmuz 2025

Tez Danışmanı: Prof. Dr. Erkay Savaş

Anahtar Kelimeler: FPGA, FHE, RNS, BFV, hızlandırıcı

HE, şifreli metinler üzerinde hesaplama yaparak verilerin gizliliğini koruyan işlemler gerçekleştirmemizi sağlar. HE kullanımı, daha ölçeklenebilir hesaplama gücüne sahip üçüncü taraflara hesaplamayı devretmek için gereken güveni ortadan kaldırabilir. HE'nin daha pratik veri işlemlerinde benimsenmesinin önündeki engel, doğasında bulunan hesaplama karmaşıklığıdır. BFV gibi mevcut HE şemaları, daha fazla uygulamanın başarım gereksinimlerini karşılamak için gevşek bağlı hızlandırıcı cihazlar aracılığıyla hızlandırılabilir. HE için bir hızlandırıcı geliştirmek, NTT gibi düzensiz bellek erişimlerine sahip algoritmaların gerçeklenmesini gerektirir. NTT algoritması hızlandırıldıktan sonra bile, ana bilgisayar ve cihaz arasındaki iletişim yükü bu hızlandırmanın bazı avantajlarını ortadan kaldırır. Daha fazla başarım artışı elde etmek için daha geniş bir aritmetik işlemler kümesinin hızlandırıcıda gerçeklenmesi gerekir. Bu tür işlemlerden biri de büyük tamsayılarla verimli işlemler gerçekleştirilmesi gerektiğinde kullanılan RNS için taban genişletmedir. Örneğin, şifre metni için kullanılan modül Q, halka boyutu $N = 2^{16}$ ile 128 bit güvenlik seviyesini karşılamak için 1747-bit olarak seçilir. RNS kullanarak, gerekli parametreleri karşılamak için 28×64-bit asal sayı veya 55×32-bit asal sayı seçebiliriz. Homomorfik çarpma sırasında, sonuçlarımız mevcut RNS tabanının sağladığı aralığa sığmayabilir. Bu durumda, temsil aralığını artırmak için işlemden önce taban genişletmesi gerekir. Kapsamlı bir HE hızlandırıcısında, NTT birimi, hızlandırıcı cihazda homomorfik çarpma işlemlerini hesaplamak için taban genişletme birimi tarafından tüketilecek kalıntılar üretir ve bunun tersi de geçerlidir. Ana bilgisayara geri iletişim kurmayarak, veri taşıma ek yükleri önlenir. Taban genişletme gerçeklememiz, cihazın programlanabilir mantık alanı için rekabet eden birden fazla

birime sahip HE hızlandırıcıları için ATP ölçütü açısından eniyilenmiştir. Cihazda bulunan bellek bant genişliği parametrelerini kullanarak hatasız taban genişletme için derleme zamanında yapılandırılabilir bir donanım üreteci sunuyoruz. Tasarım, başarımı taban genişletme aritmetiğinden ayıran ölçeklenebilir bir mimariye sahiptir. Derleme zamanında ayarlanabilir bir iş hacmi parametresi, ek mantık alanı pahasına işlemin başarımını artırır. 2^{12} ile 2^{16} arasındaki halka boyutları için FPGA kullanım sonuçlarımızı sunuyoruz. Taban genişletme mimarimizi literatürde bulunan mimarilerle karşılaştırıyoruz. En son teknolojiye sahip açık kaynaklı HE yazılım kütüphanesi OpenFHE ile FPGA gerçeklememizi karşılaştırıyoruz ve gerçeklememizin yazılım gerçeklemesine göre $\times 10 - 17$ hız artışı sağladığını gösteriyoruz.

ACKNOWLEDGEMENTS

Like many before me, I am fortunate to stand on the shoulders of giants. I want to express my heartfelt gratitude to my advisor, Professor Erkay Savaş. His attention to detail, along with his expertise and patience, helped me to understand, approach, and ultimately solve complex problems. I am particularly grateful for his support and understanding in such an early stage of my education. I wouldn't have been able to work on the exciting problems I had before joining his group.

I express my gratitude to Asst. Prof. Erdinç Öztürk for introducing me to digital design, FPGAs, and the joy of creating circuits to solve challenging problems. His unique teaching style, combined with his passion for computers, enchanted me during my bachelor's studies.

I want to thank the research group at ISEC, TU Graz. I want to thank Aikata, Anisha Mukherjee, David Jacquemin, Florian Hirner, and Florian Krieger for their friendship and helpful discussions.

I thank Prof. Ingrid Verbauwhede for the opportunity to participate in an internship at COSIC, KU Leuven. The excellent researchers at COSIC, Dr. Furkan Turan, Robin Geelen, Jonas Bertels, and Xander Pottier, made us feel welcome.

I would also like to thank my jury committee members, Prof. Özcan Öztürk and Assoc. Prof. Sujoy Sinha Roy for assessing the quality of my thesis and allowing me to improve its clarity and impact.

Thinking critically about my research has allowed me to justify and evaluate my work. Prof. Elisabeth Oswald has helped us evaluate our work by preparing study sessions and encouraging thought-stimulating discussions. I thank her for her valuable time.

Experienced researchers, such as Dr. Ahmet Can Mert, Can Ayduman, and Dr. Tolun Tosun, have guided me through the challenges of navigating research. Dr. Mert has been a trailblazer for our group, paving the way for researchers who follow in his footsteps. His sincere dedication to research is truly influential. I have immense gratitude to Can Ayduman for dedicating his valuable hours to us, debugging designs, and serving as a constant source of inspiration. His determination has not only encouraged me to overcome complex problems but also to seek them actively. Dr. Tosun has taught me how to approach engineering problems method-

ically. His work ethic and professionalism motivate the people working with him to strive for their best.

I would like to extend my gratitude to my peers and fellow researchers at CISEC, Sabancı University, for the enjoyable years I spent there, Ali Şah Özcan, Dr. Arsalan Javeed, Dr. Atıl Utku Ay, Ceren Yıldırım, Efe İzbudak, Emre Koçer, Enes Recep Türkoğlu, Dr. Kübra Kaytancı, Dr. Melik Yazıcı, and Yusuf Sür.

My parents taught me to value knowledge and education. Their tireless support and patience have allowed me to pursue this degree. They have instilled in me the resilience necessary to endure challenging situations and overcome obstacles. My father has allowed me to pursue my own path and supported me every step of the way. My mother has prepared me to meet the challenges I encounter today. I am deeply humbled by their faith in me.

Finally, I want to mention the single most important person in my life. My significant other, best friend, and lab partner, Eda. She is the toughest person I know. I cherish every moment with you. I will support you just as you supported me through everything. I am fortunate to have met you. Your love is the utmost joy I find in life. It is my hope to stand by your side through every step of our lives.

This thesis has been kindly supported by TÜBİTAK under grant number 122E222.

Hayatımın anlamına...

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATONS

ASIC Application Specific Integrated Circuit 1, 2, 9, 41, 52

ATP Area-Time Product ..... v, vii, 24, 47, 48, 51

BEHZ Bajard-Eynard-Hasan-Zucca 9, 47

BFV Brakerski-Fan-Vercauteren ..... iv, vi, 3, 4, 7, 8, 9, 22

BGV Brakerski-Gentry-Vaikuntanathan 7, 22

CC clock cycle 30

CGGI Chillotti-Gama-Georgieva-Izabachène 7, 22

CKKS Cheon-Kim-Kim-Song 4, 7, 22

CPU Central Processing Unit 9, 18, 27, 49

CRT Chinese Remainder Theorem 12, 13, 18

DFT Discrete Fourier Transform 11

DM Ducas-Micciancio 7, 22

DSP Digital Signal Processing 27, 29, 42

FHE Fully Homomorphic Encryption 1, 2, 3, 6, 7, 17, 21, 24, 37

FIFO First In First Out 25

FPGA Field Programmable Gate Array v, vii, xiv, 1, 2, 3, 4, 5, 9, 14, 18, 20, 21, 24, 25, 26, 27, 29, 32, 35, 39, 41, 42, 44, 51, 52

GPU Graphics Processing Unit 1, 27, 42

HBM High Bandwidth Memory ..... xiv, 2, 25, 26, 35

HE Homomorphic Encryption iv, v, vi, vii, xiv, 1, 2, 3, 4, 5, 6, 7, 8, 10, 21, 22, 23, 48

HPS Halevi-Polyakov-Shoup 9

LMKCDEY Lee-Micciancio-Kim-Choi-Deryabin-Eom-Yoo 7, 22

LUT Look-Up Table 27, 29

LWE Learning With Errors 7

MC Memory Controller 25

MS Mini Switch 25

NTT Number Theoretic Transform . . . . . iv, vi, 1, 11, 17, 24, 29, 32, 37, 39, 51

PC Pseudo Channel 25

PHE Partially Homomorphic Encryption 6

RLWE Ring-Learning With Errors 1, 7

RNS Residue Number System . iv, vi, 2, 3, 4, 7, 8, 9, 11, 18, 19, 20, 25, 32, 39, 42

RSA Rivest–Shamir–Adleman 4

SWHE Somewhat Homomorphic Encryption 6, 7

VLSI Very-large-scale Integration 41

1. INTRODUCTION

As the demand for processing data increases, the need to secure data and the ability to offload this data processing become increasingly relevant. Currently, data is being secured in transit via public-key cryptosystems. However, attacks on data at rest, where it is in plaintext, are a more lucrative target for malicious actors. Traditional cryptosystems do not provide security guarantees in such threat models. Thus, accessing, processing, and securing data simultaneously is an important task (Defense Advanced Research Projects Agency (2021)). To alleviate this vulnerability, HE provides a unique ability to process data without gaining knowledge of its contents. It was first conceptualized by Rivest et al. (1978). The first Fully Homomorphic Encryption (FHE) scheme was introduced by Gentry (2009), based on the ideal coset problem now considered insecure, which used the "bootstrapping" operation to achieve the full homomorphism. Later schemes utilized the Ring-Learning With Errors (RLWE) problem, which is still in use today. Hence, the schemes currently rely on operations with integer polynomials with a degree $N-1$ where the practical range is $2^{12} < N < 2^{16}$.

The current implementation of FHE requires computationally expensive operations such as NTT, base extensions found in homomorphic multiplication, and relinearization. Homomorphic multiplication and relinearization enable us to multiply two encrypted ciphertexts, each comprising two polynomial terms, and are fundamental operations for realizing privacy-preserving machine learning applications. NTT is used for improving the computational complexity of polynomial multiplication found in homomorphic multiplication within FHE. It reduces the complexity from schoolbook approach $O(N^2)$ to $O(N \log N)$. NTT has irregular memory accesses, and throughput requirements cannot be met utilizing traditional software approaches. Thus, there are works on accelerating the computationally expensive operations on accelerators such as Graphics Processing Unit (GPU) (Badawi et al. (2018), Özcan et al. (2023)), FPGA (Mert et al. (2022), Agrawal et al. (2023)), and Application Specific Integrated Circuit (ASIC) (Samardzic et al. (2021)) devices. Although GPUs have become prevalent due to their ubiquitous presence in accelerating artifi-

cial intelligence workloads, their main performance advantages are again specialized for artificial intelligence workloads. The low-precision floating-point operations utilized in artificial intelligence workloads do not directly translate into performance benefits for FHE use cases. ASICs on the other hand, are specifically designed for accelerating FHE workloads, which provides a wide design space for acceleration. However, the state-of-the-art FHE parameters and use cases are still evolving. Hence, the lead time for an ASIC may be disadvantageous for use in FHE research. FPGAs provide a middle ground between these options. Their reconfigurability allows for relatively fast prototyping and experimentation, but their performance is limited by the amount of input-output bandwidth to and from the FPGA device.

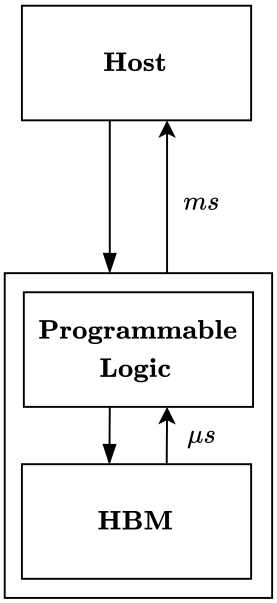


Figure 1.1 Memory hierarchy for the accelerator

To overcome this limitation of FPGA devices, we aim to reduce the communication overhead between the host and the device. We choose a data center acceleration card from AMD, which features a HBM off-chip memory coupled to the FPGA fabric. HBM provides high input/output bandwidth storage, which reduces the impact of memory read/write bottlenecks compared to traditional DRAM. It allows us to store the coefficients required to compute FHE on the device. Accessing the HBM memory is faster than copying data from the host to the device and vice versa. We make use of this memory hierarchy shown in Figure 1.1 to fit as many functions on the device itself.

The current generation of HE schemes usually requires large integer arithmetic. The required modulus for arithmetic may be as large as a thousand bits. Implementing this arithmetic naively is not practical. The state-of-the-art methods utilize RNS arithmetic to handle these large integers. RNS representation allows us to work with

machine word-size arithmetic and parallelize computations. One such operation is the homomorphic multiplication. Homomorphic multiplication, together with relinearization, consumes a significant amount of time in HE computations (Halevi et al. (2018)). In homomorphic multiplication, we are multiplying two ciphertexts, each containing two polynomials of degrees in the range of $2^{12} \leq N \leq 2^{16}$ that are represented in a large RNS modulus Q with l many $\lceil \log_2(q_i + 1) \rceil -$bit residues. However, when the results of an operation do not fit within the provided precision, a base extension operation must be performed before the operation.

Base extension operations require reading the source residues, remainders in coprime moduli, and performing operations to get the results in the target moduli. Hence, it is a memory-intensive operation. For 64-bit residues, an example parameter setting would be to read 18 residue polynomials, each with size $N = 2^{16}$ and produce 17 residue polynomials . Our work aims to provide a flexible and scalable solution for performing base extensions by leveraging the parallel nature of FPGAs. We outline a specific algorithm implementation in a relinearization operation accelerator for FHE. We focus on the Shenoy and Kumaresan (1989) exact base extension algorithm implementation on FPGA.

1.1 Contributions

Our goal is to accelerate homomorphic multiplication and relinearization operations of BFV (Brakerski (2012), Fan and Vercauteren (2012)). By implementing base extension functionality on a FPGA loosely coupled accelerator hardware, we avoid costly transfers between the host and the device for homomorphic multiplication and relinearization. This design choice theoretically improves end-to-end acceleration times on the order of milliseconds. Our contributions in this thesis are as follows.

- To the best of our knowledge, this is the first configurable exact polynomial base extension architecture for FHE use on FPGA.
- We improve the overall runtime of FHE accelerators by staying on the device for a longer period. Implementing more functionality on the device itself.
- We implement the base extension algorithm optimized for use with polynomials in RNS domain.
- We provide a scalable parametric base extension hardware generator for

FPGA, where we are limited by the throughput of the available interface.

- We optimize our hardware architecture in terms of area-time product, which is necessary for the rest of the HE acceleration functions to fit to the FPGA programmable logic.
- We provide a base extension architecture where memory requirements grow linearly with the number of target bases we are extending to. Our memory requirements do not change noticeably with the number of source bases.
- We measure our implementation against the state-of-the-art HE library OpenFHE and show that we are consistently an order of magnitude faster with the given parameters.

1.2 Relevant Works

There are existing works that primarily focus on public key cryptography applications where representations have a low number of source and target base residues. Early works with applications in cryptography have focused on the Rivest–Shamir–Adleman (RSA) algorithm, specifically modular exponentiation. The work by Kawamura et al. (2000) serves as the foundation for accelerating base extensions in RSA. Their selection of parameters for correctness guarantees allows them to reduce the complexity of their circuit. This selection, however, limits the number of available primes that can be used in the cryptosystem. For HE schemes, this hinders their usability due to the lack of options for the RNS representation.

The current implementations of base extension for use in HE utilize the Fast Base Conversion algorithm, which actually computes an approximation of the base extension rather than an exact one. This approach is utilized and accommodated in the Cheon-Kim-Kim-Song (CKKS) (Cheon et al. (2017)) scheme, but introduces errors that are incompatible with the BFV scheme, which cannot recover from these errors. Works such as Halevi et al. (2018) that target BFV (Brakerski (2012), Fan and Vercauteren (2012)), make use of floating-point arithmetic for computing the base extension without error but are limited by precision and availability of floating-point units. Works that utilize fixed-point approximations for floating-point arithmetic exist, but the circuit complexity for computing this approximation is high (Turan et al. (2020), Su et al. (2022), Van Beirendonck et al. (2023)). Lack of configurability

and scalability among the works has also widened the research gap in the literature.

Work by Badawi et al. (2018) utilizes mixed-radix conversions, which do not require any floating-point operations or approximations. However, we justify that on devices with limited input-output bandwidth, such as FPGAs, the mixed-radix conversion is difficult to implement due to its sequential nature and quadratic complexity for a pipelined implementation.

1.3 Thesis Outline

The thesis is organized as follows .

- In Chapter 2, we introduce the background and notation necessary for understanding the contributions being made in the thesis.
- In Chapter 3, we present our work and methodology in achieving the results. We present insights into the design decisions being made.
- In Chapter 4, we present the FPGA utilization results for various parameter sets and discuss their implications. We also present a comparison with the state-of-the-art HE library OpenFHE for context.
- In Chapter 5, we present insights resulting from our work and suggest future research directions.

2. BACKGROUND

In this section, we will provide the necessary background for understanding the contributions made in this thesis. For simplicity, we will be giving overviews of only the parts that are relevant to our work.

2.1 Notation

$\mathbf{T}(x)$ represents a polynomial in $\mathcal{R}_Q$, where $\mathcal{R}_Q$ represents the ring $\mathbb{Z}_Q/(x^N+1)$, and $Q = \prod_{i=0}^{l} q_i$. A polynomial residue of $\mathbf{T}(x)$ in q_i is shown as $\mathbf{t}_i(x)$ and the coefficient with power r, $t_{i_r} \cdot x_i^r$, is shown as t_{i_r}. Individual digits of an integer are shown as $a_{\{i\}}$, with the base stated within the context. For example, in base-2 let $a = 1110011$ then $a_{\{4\}} = 1$.

2.2 Homomorphic Encryption

HE allows us to perform processing on ciphertexts. Once decrypted, the results are the same as if we had performed the operation on plaintexts. Thus, eliminates the trust needed to delegate the computation to a third party. An example computation is shown in Figure 2.1.

There are three types of HE: Partially Homomorphic Encryption (PHE), Somewhat Homomorphic Encryption (SWHE), and FHE. PHE only allows additive or multiplicative operations and has very limited applicability . SWHE allows both additive and multiplicative operations, but is limited in terms of operation depth. FHE

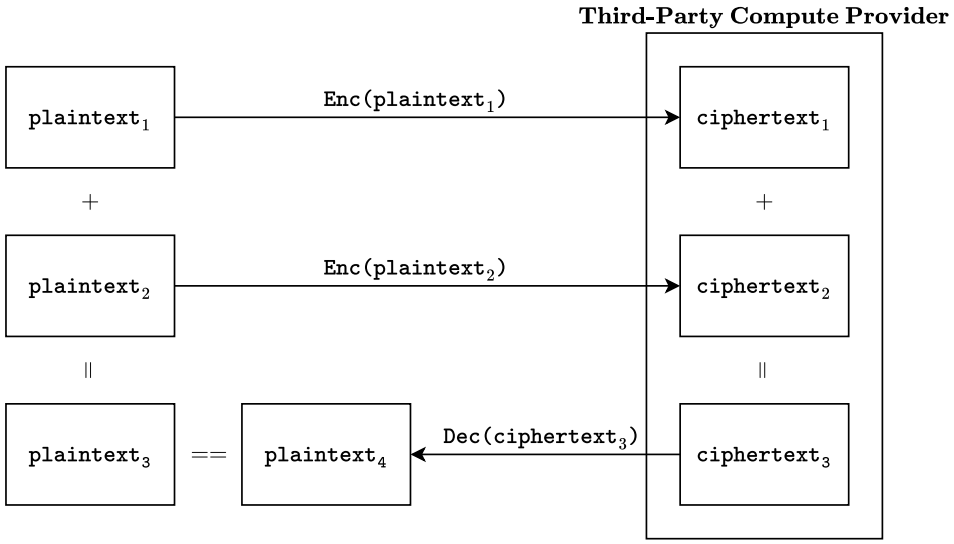


Figure 2.1 Example HE computation

allows us to perform an arbitrary number of operations of both additive and multiplicative types. FHE is usually achieved by performing a "bootstrapping" operation on SWHE to reset the noise on the ciphertext.

There are several HE schemes in the literature. We will make a distinction between two scheme categories. We follow the categorization of the OpenFHE (Al Badawi et al. (2022)) library. First is Brakerski-Gentry-Vaikuntanathan (BGV)-like schemes; BFV, CKKS. Second, Ducas-Micciancio (DM)-like schemes, Chillotti-Gama-Georgieva-Izabachène (CGGI), Lee-Micciancio-Kim-Choi-Deryabin-Eom-Yoo (LMKCDEY). Our contributions target the BGV-like schemes, specifically BFV.

2.2.1 BFV Scheme

The BFV scheme (Fan and Vercauteren (2012)) builds upon the work of Brakerski (2012) by adapting the underlying arithmetic from Learning With Errors (LWE) to RLWE. It is particularly targeting integer operations with representation limited by the plaintext modulus. It does not induce additional constraints regarding the moduli required to implement the scheme, given that the security conditions are met, unlike BGV and CKKS. However, current bootstrapping approaches for BFV are not as practical as for CKKS.

We will consider the RNS version of the BFV scheme and give an overview of the two methods for implementing it. We follow the notations of Kim et al. (2021). Secret key $\mathtt{sk}$ is sampled from the distribution $\chi_{\mathtt{k}}$, $\{-1, 0, 1\}$ uniformly randomly.

Error e is sampled from a discrete Gaussian distribution χ_e. Otherwise, coefficients are sampled from $\mathcal{R}_Q$ uniformly. Plaintext modulus is t and $\Delta = \lfloor \frac{Q}{t} \rfloor$. The basic operations for the BFV HE scheme are as follows.

- $\texttt{SKeyGen}_{\texttt{BFV}}()$: $\texttt{sk} \xleftarrow{\$} \chi_{\texttt{k}}$
- $\texttt{PKeyGen}_{\texttt{BFV}}(\texttt{sk})$: $\mathbf{a} \xleftarrow{\$} \mathcal{R}_{\mathcal{Q}}, \quad \mathbf{e} \xleftarrow{\$} \chi_e, \quad \texttt{pk} \leftarrow (|\mathbf{a} \cdot \texttt{sk} + \mathbf{e}|_Q, -a) \in \mathcal{R}_Q^2$
- $\texttt{EvalKeyGen}_{\texttt{BFV}}(\texttt{sk}, T)$: $\texttt{rlk} \leftarrow (\left|-(\mathbf{a} \cdot \texttt{sk} + \mathbf{e}_i) + T^i \cdot \texttt{sk}^2\right|_Q, \mathbf{a}_i)$ where $i \in [0, \lfloor \log_T Q \rfloor]$, T is the base for decomposition. The method utilized here is referred to as version 1 in Fan and Vercauteren (2012).
- $\texttt{Encrypt}_{\texttt{BFV}}(\texttt{pk}, \texttt{pt})$: $\texttt{ct} \leftarrow (|\texttt{pk}_0 \cdot \mathbf{u} + \mathbf{e}_1 + \Delta \cdot \texttt{pt}|_Q, |\texttt{pk}_1 \cdot \mathbf{u} + \mathbf{e}_2|_Q)$ where $\texttt{pt} \in \mathcal{R}_t, \quad \mathbf{u} \xleftarrow{\$} \chi_k, \quad \mathbf{e}_i \xleftarrow{\$} \chi_e$
- $\texttt{Decrypt}_{\texttt{BFV}}(\texttt{sk}, \texttt{ct})$: $\texttt{pt} \leftarrow \left| \lfloor \frac{t \cdot |\texttt{ct}_0 + \texttt{ct}_1 \cdot \texttt{sk}|_Q}{Q} \rceil \right|_t$
- $\texttt{Add}_{\texttt{BFV}}(\texttt{ct}_1, \texttt{ct}_2)$: $(|\texttt{ct}_{1_0} + \texttt{ct}_{2_0}|_Q, |\texttt{ct}_{1_1} + \texttt{ct}_{2_1}|_Q)$
- $\texttt{Multiply}_{\texttt{BFV}}(\texttt{rlk}, \texttt{ct}_1, \texttt{ct}_2)$:

$$\texttt{ct}_\star = \texttt{ct}_{\star_0} \leftarrow \left| \lfloor \frac{t \cdot (\texttt{ct}_{1_0} \cdot \texttt{ct}_{2_0})}{Q} \rceil \right|_Q,$$

$$\texttt{ct}_{\star_1} \leftarrow \left| \lfloor \frac{t \cdot (\texttt{ct}_{1_0} \cdot \texttt{ct}_{2_1} + \texttt{ct}_{1_1} \cdot \texttt{ct}_{2_0})}{Q} \rceil \right|_Q,$$

$$\texttt{ct}_{\star_2} \leftarrow \left| \lfloor \frac{t \cdot (\texttt{ct}_{1_1} \cdot \texttt{ct}_{2_1})}{Q} \rceil \right|_Q$$

Then relinearize as follows, where $\texttt{ct}_{\star_2} = \sum_{i=0}^{\lfloor \log_T Q \rfloor} \texttt{ct}_{\star_{2_{\{i\}}}} T^i$,

$$\texttt{ct}_{\texttt{mult}_1} \leftarrow \left| \texttt{ct}_{\star_0} + \sum_{i=0}^{\lfloor \log_T Q \rfloor} \texttt{rlk}_{i_0} \cdot \texttt{ct}_{\star_{2_{\{i\}}}} \right|_Q,$$

$$\texttt{ct}_{\texttt{mult}_2} \leftarrow \left| \texttt{ct}_{\star_1} + \sum_{i=0}^{\lfloor \log_T Q \rfloor} \texttt{rlk}_{i_1} \cdot \texttt{ct}_{\star_{2_{\{i\}}}} \right|_Q$$

As with most HE schemes, BFV also has RNS versions. Specifically, there are two RNS variants of BFV.

2.2.1.1 BEHZ Method

The first RNS version of BFV is by Bajard et al. (2017), referred to as Bajard-Eynard-Hasan-Zucca (BEHZ), which introduces RNS versions of homomorphic multiplication and decryption. It proposes methods to perform the division and rounding steps, called `DR` in the paper, which are traditionally incompatible with RNS representation, as it is a non-positional number system. BEHZ method presents RNS compatible versions of the `DR` method, they introduce an approximate `DR` which can be computed in RNS.

The algorithm shown in Figure 2.2 closely follows the notation used in OpenFHE implementations. For the actual algorithm, please refer to Bajard et al. (2017). We will be utilizing the BEHZ method for our homomorphic multiplication acceleration.

Our contribution is to implement and accelerate the "BaseConvSK" block shown in Figure 2.2. Where the source base is $\mathcal{B}_{\mathtt{sk}}$ and the target base is Q. $\mathcal{B}_{\mathtt{sk}}$ is the auxiliary basis comprised of $\mathcal{B}$ and $m_{\mathtt{sk}}$. Note that $m_{\mathtt{sk}}$ is our redundant residue that will be consumed by Algorithm 2. Unlike the Halevi-Polyakov-Shoup (HPS) method, it can be implemented utilizing purely integer arithmetic, which increases its suitability for FPGA acceleration.

2.2.1.2 HPS Method

Following on the BEHZ method, the work by Halevi et al. (2018) introduced the HPS variant. Claiming improved and simpler routines for computing base extensions, which introduce less noise than BEHZ. The claim has been challenged by Bajard et al. (2019), arguing that the implementations do not follow the paper. Currently, the software implementation of HPS is slightly faster than the BEHZ method in the OpenFHE library. It utilizes floating-point arithmetic for computing base extensions and reduces the number of integer multiplications by shifting some of the integer arithmetic load to floating-point arithmetic. The authors mention the works by Shenoy and Kumaresan (1989) and Kawamura et al. (2000) and claim that these works increase the number of integer operations significantly. For Central Processing Unit (CPU) implementations, where floating-point units are ubiquitous, this claim may be conclusive. Still, on platforms where floating-point arithmetic usage costs more logic area than the integer counterparts, such as FPGAs, ASICs, this method becomes less practical. Their timing results show that with increasing

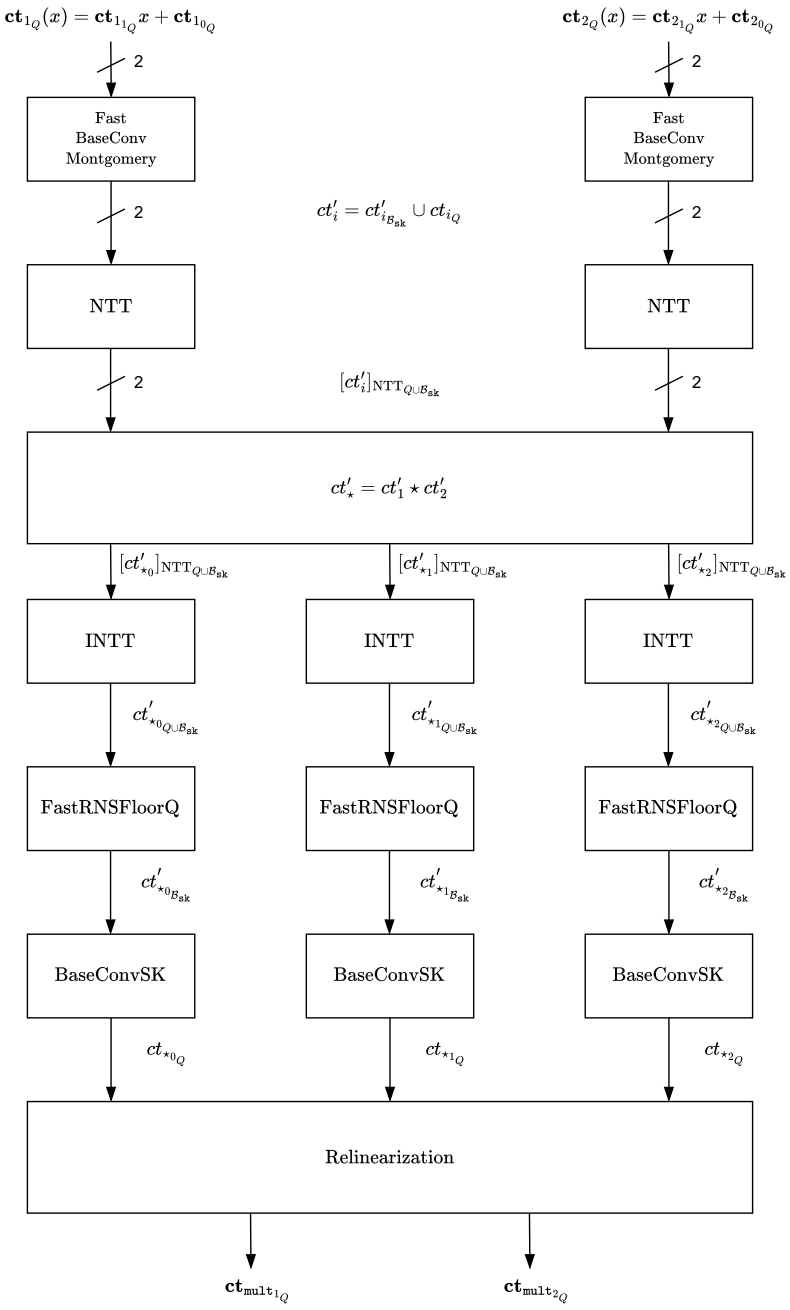


Figure 2.2 Homomorphic multiplication in BEHZ method for BFV HE scheme Bajard et al. (2017) adapted from Mert (2021)

sizes of modulus Q, the base extension consumes a non-negligible amount of time while computing homomorphic multiplications.

2.3 Residue Number System

Here we introduce an overview of the RNS, a non-positional representation of integers. Positional representations are utilized in schoolbook arithmetic where the radices are in the powers of the radix. It is straightforward to compare two numbers in the same positional representation. However, in RNS we do not have ordered positional radices. Each radix, or residue, is the remainder of the division of the original integer by a coprime integer. The multiplication of these coprime integers, usually a smooth number, forms our representation range. Using the following theorem, we see that this representation exists and is unique.

Theorem 2.1 (Chinese Remainder Theorem)**.** Given l coprime integers $q = (q_0, \ldots, q_{l-1})$ and their product $Q = \prod_{i=0}^{l-1} q_i$, consider $a_i \in (a_0, \ldots, a_{l-1}) \mid a_i < q_i$. Then,

$$\exists A \mid 0 \leq A < Q, \quad a_i \equiv A \pmod{q_i}, \quad \forall i \in 0 \leq i \leq l-1 \tag{2.1}$$

The factors of Q are generally chosen to be within machine word sizes. This design choice allows us to perform arithmetic on large integers by working with smaller, word-sized integers. For our use case, these word sizes will be either 32 or 64 bits.

2.4 Number Theoretic Transform

NTT is Discrete Fourier Transform (DFT) performed over a ring, specifically $\mathcal{R}_Q$ where N is a power of two . Due to the modulus Q being in RNS representation, our NTTs are defined over factors of Q, denoted as q_i. We will assume this condition on the polynomial degree for the rest of the thesis. We will also assume that we will work with the nega-cyclic convolution variant of the NTT. To perform NTT, we have to first choose prime moduli with the following condition for the existence of $2N-$th root of unity.

$$q \equiv 1 \pmod{2N} \tag{2.2}$$

Then the $2N-$th root of unity with the following condition is guaranteed to exist.

$$\begin{aligned} \psi^{2N} &\equiv 1 \pmod{q} \\ \psi^{s} &\not\equiv 1 \pmod{q} \quad | \quad s \in [1, 2N-1] \end{aligned} \tag{2.3}$$

2.5 Base Extension

There are two approaches to computing target base residues from the source base residues. The first way is to reconstruct the number using Chinese Remainder Theorem (CRT). The second way to compute residues is by using Mixed-Radix Conversion.

2.5.1 CRT-Based Algorithms

Given the residues a_{q_i}, we can reconstruct the full integer A utilizing the following formula.

$$A = \left| \sum_{i=0}^{l-1} \left| a_{q_i} \cdot \hat{Q}_i^{-1} \right|_{q_i} \cdot \hat{Q}_i \right|_Q \tag{2.4}$$

Where, $\hat{Q}_i = \frac{Q}{q_i}$. This formula has the crucial limitation of requiring a modulo operation by the full precision Q integer, which can be represented in the following equation.

$$A = \left(\sum_{i=0}^{l-1} \left| a_{q_i} \cdot \hat{Q}_i^{-1} \right|_{q_i} \cdot \hat{Q}_i \right) - \alpha \cdot Q \tag{2.5}$$

To compute new residues exactly from the existing base, we have to calculate the

α term, where α is in the range $[0, l-1]$. In the literature, methods exist that compute α by successively computing its binary representation or using floating-point arithmetic and rounding operations.

Algorithms based on CRT involve computing cascaded summation operations and can be easily pipelined. However, there needs to be a bound on the input or an extra residue to compute the new residues without reconstructing the full precision number.

2.5.1.1 Fast Base Conversion Algorithm

If the errors in the produced representation can be tolerated, we can skip computing α to reduce design and computation complexity. Fast Base Conversion, shown in Algorithm 1, utilizes the CRT Equation 2.5 without the $-|\alpha \cdot Q|_{p_j}$ computation .

$$a'_{p_j} = \left| \sum_{i=0}^{l-1} \left| a_{q_i} \cdot \hat{Q}_i^{-1} \right|_{q_i} \cdot \hat{Q}_i \right|_{p_j} \tag{2.6}$$

Algorithm 1 Fast Base Conversion Algorithm

Input: $A_Q = a_{q_i} (\forall i), A < Q;\ Q = \{q_0, \ldots, q_{l-1}\},\ P = \{p_0, \ldots, p_{k-1}\}$
Precompute: $\left|\hat{Q}_i^{-1}\right|_{q_i}, \left|\hat{Q}_i\right|_{p_j}, \left|\hat{Q}_i\right|_{q_l}, \left|Q^{-1}\right|_{q_l}, |Q|_{p_j} (\forall i, j)$
Output: $A'_P = a'_{p_j} (\forall j), a'_{p_j} < a_{p_j} + \alpha \cdot Q \pmod{p_j}$

1: $\mu_{q_i} \leftarrow a_{q_i} \cdot \left|Q_i^{-1}\right|_{q_i} \pmod{q_i} (\forall i)$
2: $\mu_{p_{j,i}} \leftarrow \mu_{q_i} \cdot \left|\hat{Q}_i\right|_{p_j} \pmod{p_j} (\forall i, j)$
3: $a'_{p_j} \leftarrow \sum_{i=0}^{l-1} \mu_{p_{j,i}} \pmod{p_j} (\forall j)$

Due to not computing the actual result by skipping the α computation, the result may contain errors.

2.5.1.2 Shenoy-Kumaresan Algorithm

In this method of calculating base extension Shenoy and Kumaresan (1989), we utilize the CRT algorithm directly. To calculate the new residue without calculat-

ing the full precision number, a redundant residue $a_{q_l} \equiv |A|_{q_l}$, such that $q_l > l$, is calculated alongside the regular source residues.

$$a_{q_l} \equiv \left| \left| \sum_{i=0}^{l-1} \left| a_{q_i} \cdot \hat{Q}_i^{-1} \right|_{q_i} \cdot \hat{Q}_i \right|_{q_l} - |\alpha \cdot Q|_{q_l} \right|_{q_l} \tag{2.7}$$

Adding $|\alpha \cdot Q|_{q_l}$ to both sides yields the following.

$$a_{q_l} + |\alpha \cdot Q|_{q_l} \equiv \left| \sum_{i=0}^{l-1} \left| a_{q_i} \cdot \hat{Q}_i^{-1} \right|_{q_i} \cdot \hat{Q}_i \right|_{q_l} \tag{2.8}$$

Then the term α becomes,

$$|\alpha|_{q_l} \equiv \left| Q^{-1} \right|_{q_l} \cdot \left| \left| \sum_{i=0}^{l-1} \left| a_{q_i} \cdot \hat{Q}_i^{-1} \right|_{q_i} \cdot \hat{Q}_i \right|_{q_l} - a_{q_l} \right|_{q_l} \tag{2.9}$$

since we know $\alpha < q_l$, this also becomes $|\alpha|_{q_l} = \alpha$.

Algorithm 2 Shenoy and Kumaresan (1989) Algorithm

Input: $A_{Q \cup q_l} = a_{q_i}(\forall i), A < Q$; $Q = \{q_0, \ldots, q_{l-1}\}$, $P = \{p_0, \ldots, p_{k-1}\}$
Precompute: $\left|\hat{Q}_i^{-1}\right|_{q_i}, \left|\hat{Q}_i\right|_{p_j}, \left|\hat{Q}_i\right|_{q_l}, \left|Q^{-1}\right|_{q_l}, |Q|_{p_j} (\forall i, j)$
Output: $A_P = a_{p_j}(\forall j)$

1: $\mu_{q_i} \leftarrow a_{q_i} \cdot \left|\hat{Q}_i^{-1}\right|_{q_i} \pmod{q_i}(\forall i)$
2: $\mu_{p_{j,i}} \leftarrow \mu_{q_i} \cdot \left|\hat{Q}_i\right|_{p_j} \pmod{p_j}(\forall i, j)$
   $\mu_{q_{l,i}} \leftarrow \mu_{q_i} \cdot \left|\hat{Q}_i\right|_{q_l} \pmod{q_l}(\forall i)$
3: $a'_{p_j} \leftarrow \sum_{i=0}^{l-1} \mu_{p_{j,i}} \pmod{p_j}(\forall j)$
   $a'_{q_l} \leftarrow \sum_{i=0}^{l-1} \mu_{q_{l,i}} \pmod{q_l}$
4: $ms_{q_l} \leftarrow a'_{q_l} - a_{q_l} \pmod{q_l}$
5: $\mu'_{q_l} \leftarrow ms_{q_l} \cdot \left|Q^{-1}\right|_{q_l} \pmod{q_l}$ ▷ μ'_{q_l} *is* α
6: $\mu'_{p_j} \leftarrow \mu'_{q_l} \cdot |Q|_{p_j} \pmod{p_j}(\forall j)$
7: $a_{p_j} \leftarrow a'_{p_j} - \mu'_{p_j} \pmod{p_j}(\forall j)$

Here we notice that Algorithm 2 lends itself well to parallelism. We will base our FPGA hardware accelerator implementation on this algorithm. The algorithm con-

sists of integer arithmetic. The values that cannot be computed on the device but are necessary for the algorithm computation can be loaded to the device after being precomputed.

2.5.1.3 Kawamura et al. Algorithm

Another approach to compute the α without keeping redundant residues is using Kawamura et al. (2000). Algorithm 3 works by recursively computing the individual bits of α given that the input range is within the error bounds.

We will derive the Kawamura et al. (2000) base extension from Equation 2.5. First, we will define ξ_{q_i} as follows.

$$\xi_{q_i} = \left| a_{q_i} \cdot \hat{Q}_i^{-1} \right|_{q_i} \tag{2.10}$$

Our Equation 2.5 becomes as follows.

$$A = \left(\sum_{i=0}^{l-1} \xi_{q_i} \cdot \hat{Q}_i \right) - \alpha \cdot Q \tag{2.11}$$

To get the α term, we divide the equation by Q.

$$\frac{A}{Q} = \left(\sum_{i=0}^{l-1} \frac{\xi_{q_i}}{q_i} \right) - \alpha \tag{2.12}$$

If we group together the unknowns α and A (which is a large integer) the equation becomes.

$$\alpha + \frac{A}{Q} = \sum_{i=0}^{l-1} \frac{\xi_{q_i}}{q_i} \tag{2.13}$$

Then we know $0 \leq \frac{A}{Q} < 1$ and $\alpha \leq \sum_{i=0}^{l-1} \frac{\xi_{q_i}}{q_i} < \alpha + 1$. So we can conclude the following.

$$\alpha = \left\lfloor \sum_{i=0}^{l-1} \frac{\xi_{q_i}}{q_i} \right\rfloor \tag{2.14}$$

Still, this computation cannot be easily done on hardware. To compute this division using integer fixed-point arithmetic, it is best to avoid division by an arbitrary integer. Hence, the division by q_i is approximated by 2^r, which is the bit-width of modulus . This division is a trivial shift operation on hardware and is essentially performed at no cost. To simplify the hardware for the division, the numerator ξ_{q_i} is also represented by its most significant ϱ bits, which are smaller than the full precision r bits. The hardware optimized equation for Equation 2.14 becomes as follows.

$$\alpha' = \left\lfloor \sum_{i=0}^{l-1} \frac{\mathbf{trunc}(\xi_{q_i})}{2^r} + \delta \right\rfloor. \tag{2.15}$$

The δ term is the initial offset to compute the flooring operation correctly. The δ value will affect the input range and will be determined by errors introduced by approximating ξ_{q_i} and q_i. Then, the approach is to compute α in a recursive manner. Computing each bit in each iteration as follows

$$\begin{aligned} \sigma_{-1} &= \delta, \\ \sigma_i &= \sigma_{i-1} + \frac{\mathbf{trunc}(\xi_{q_i})}{2^r} \\ \alpha_{\{i\}} &= \lfloor \sigma_i \rfloor \\ \sigma_i &= \sigma_i - \alpha_{\{i\}} \quad (\forall i \in [0, l-1]). \end{aligned} \tag{2.16}$$

The errors introduced by these two approximations are calculated to determine the error-free operation range. First, the error by approximating the division by q_i with 2^r is calculated as follows.

$$\begin{aligned} \epsilon_{q_i} &= \frac{2^r - q_i}{2^r} \quad (\forall i) \\ \epsilon_Q &= \mathbf{max}(\epsilon_{q_i}) \end{aligned} \tag{2.17}$$

Second, the error by truncating the value of ξ_{q_i} becomes the following.

$$\lambda q_i = \frac{\xi_{q_i} - \mathbf{trunc}(\xi_{q_i})}{q_i} \quad (\forall i)$$
$$\lambda_Q = \mathbf{max}(\lambda_{q_i}) \tag{2.18}$$

The base extension operation will compute the new residues without error given that $0 \leq l \cdot (\epsilon_Q + \lambda_Q) < 1$ and $0 \leq A < (1-\delta) \cdot M$. So, the approximations reduce the range of the inputs. Thus, this algorithm benefits from a careful choice of parameters to function correctly. In implementing this algorithm, the value of r is usually fixed for the hardware design. So, the choice of predetermined primes close to 2^r is necessary for this algorithm to take a large range of inputs. The approximation by the use of **trunc**() in practice is not costly as it only affects the adder precision, which can be increased to accommodate range requirements.

However, this algorithm is not disadvantageous for use in FHE applications due to the choice of primes. Specifically, the NTT algorithm commonly utilized in FHE requires primes to be in the form of $q \equiv 1 \pmod{2N}$, which limits the available primes significantly. Satisfying the FHE security requirements in terms of the size of modulus requires the availability of sufficient primes, as shown by the work of Bossuat et al. (2025) .

Algorithm 3 Recursive Base Extension Algorithm (Kawamura et al. (2000))

Input: $A_Q = a_{q_i}(\forall i)$; $Q = \{q_0, \ldots, q_{l-1}\}$, $P = \{p_0, \ldots, p_{k-1}\}$
Precompute: δ, $\left|\hat{Q}_i\right|_{q_i}^{-1}$, $\left|\hat{Q}_i\right|_{p_j}(\forall i, j)$, $|-Q|_{p_j}(\forall j)$
Output: $A_P = a_{p_j}(\forall j)$

procedure TRUNC(ξ, r, ϱ)
  $\xi \leftarrow \xi \wedge ((1 \ldots 1)_{[\varrho-1:0]} \ll (r-\varrho))$ ▷ *Truncate $(r-\varrho)$ least-significant bits*
end procedure
$\xi_{q_i} \leftarrow a_{q_i} \cdot \left|\hat{Q}_i^{-1}\right|_{q_i} \pmod{q_i} \quad (\forall i)$
$\sigma_{-1} \leftarrow \delta, \quad y_{j,0} \leftarrow 0 \quad (\forall j)$
for $s = 0, \cdots, l-1$ **do**
  $\sigma_s \leftarrow \sigma_{(s-1)} + \text{TRUNC}(\xi_{q_s})/2^r$
  $\alpha_{\{s\}} \leftarrow \lfloor \sigma_s \rfloor$ ▷ $\alpha_{\{s\}} \in \{0, 1\}$
  $\sigma_s \leftarrow \sigma_s - \alpha_{\{s\}}$
  $y_{j,s} \leftarrow y_{j,(s-1)} + \xi_{q_s} \cdot \left|\hat{Q}_s\right|_{p_j} + \alpha_{\{s\}} \cdot |-Q|_{p_j}(\forall j)$
end for
$a_{p_j} \leftarrow y_{j,n} \pmod{p_j}(\forall j)$

2.5.1.4 Halevi-Polyakov-Shoup Optimization

An optimization presented in Halevi et al. (2018) utilizes the floating-point units available in CPU. This approach computes the α through the Equation 2.14 but does not approximate the ξ_{q_i}'s or the division by q_i. Instead, it uses floating-point operations to calculate the divisions and sums the result. The barrier to adopting this method directly on hardware such as the FPGA is the lack of widely available native floating-point units, and even if they are available, the non-trivial implementation of the floating-point division algorithm on the device. There are two ways to overcome this limitation: either using fixed-point approximations with precomputed values or keeping the computation for α in the RNS representation and calculating each residue separately, as shown in Sinha Roy et al. (2019). Implementing the floating-point version or the calculation by each residue in the hardware is expected to cost more programmable logic area than their integer counterparts. Lastly, the precision provided by the IEEE 754 floats and double floats to compute the rounding operation for α produces a small error term. This would increase further by the use of a fixed-point approximation.

2.5.2 Mixed-Radix-Based Algorithm

The algorithms based on CRT required a limited input range, a redundant residue, or floating-point computation to produce new residues. However, there is an alternative method to compute residues: to keep the RNS residues but convert them to a positional number system. The mixed-radix algorithm (see Algorithm 4) actually converts RNS residues, which are in a non-positional representation, to a positional representation. Then, the new residue is computed by a relation between its representations. If the computation and parameters are chosen suitably, all the operations are within the machine word size.

Given an integer B in $[0, Q)$ we can represent this integer as follows

$$B = b_{l-1} \prod_{i=0}^{l-2} R_i + \cdots + b_2 R_1 R_0 + b_1 R_0 + b_0, \tag{2.19}$$

where R_i are corresponding radices in the mixed-radix representation. b_i are the

digits such that they are less than their radices, $R_i > b_i$. The integer B in the mixed-radix representation will be presented as $\langle b_{l-1}, b_{l-2}, \ldots, b_0 \rangle$ in an ordered fashion, where the most significant digit is b_{l-1}.

To convert the RNS representation of a number into the mixed-radix system, we can simply copy the first residue from the RNS as follows.

$$b_0 = |B|_{q_0} \tag{2.20}$$

Then we can produce the rest of the mixed-radix digits as follows

$$b_{i+1} = \left| \frac{B - b_i}{q_i} \right|_{q_{i+1}} = \left| \frac{B}{\prod_{j=0}^{i} q_j} \right|_{q_{i+1}}. \tag{2.21}$$

The key idea is to compute the residue of the new prime through the conversion process. Given the target modulus q_l, which is pairwise co-prime with the previous primes, we know that the representation of B in the new mixed-radix digit is 0.

$$b_l = 0 \tag{2.22}$$

We have the residue representation, as follows

$$\{|B|_{q_0}, |B|_{q_1}, \cdots, |B|_{q_{l-1}}\}. \tag{2.23}$$

And, we want to find $|B|_{q_l}$. We adapt the first version of Equation 2.21 to compute the mixed-radix digits, as we do not have the option to divide the natural representation of B by the product of the moduli. We will include the $|B|_{q_l}$ symbolically in the computation, such that we will find the relation where we get the following equation

$$b_l = \zeta \times |B|_{q_l} + \theta = 0, \quad b_l, \zeta, \theta \in \mathbb{Z}_{q_l}. \tag{2.24}$$

Furthermore, we know the following

$$\zeta = \left| \prod_{i=1}^{l} q_i \right|_{q_l} = |Q|_{q_l}. \tag{2.25}$$

To compute θ, we will compute the mixed-radix conversion, starting with the initial value $\theta_{-1} = 0$.

Then, we can compute subsequent θ_0 utilizing the first equation.

$$\theta_0 = \left| \frac{\theta_{-1} - b_0}{q_0} \right|_{q_l} \tag{2.26}$$

More generally,

$$\theta_i = \left| \frac{\theta_{i-1} - b_i}{q_i} \right|_{q_l} \tag{2.27}$$

We will compute mixed-radix digits b_i's using Equation 2.21 using the first equality. So, to rewrite Equation 2.21 for our use case, we obtain the following

$$b_{i+1} = \left| \frac{|B|_{q_i} - b_i}{q_i} \right|_{q_{i+1}}. \tag{2.28}$$

Finally, we get the last θ and we are able to plug this value back to Equation 2.24 to compute $|B|_{q_l}$

$$\theta = \theta_{l-1}. \tag{2.29}$$

Notice that for the hardware implementation, it is important that q_i's are computed in an ordered manner, such as the following

$$q_0 < q_1 < \cdots < q_{l-1} < q_l. \tag{2.30}$$

Due to the fact that we do not know the full precision B, but the RNS representation of B, we will have to subtract b_i's per radix computation iteration. For example, to compute b_2 on hardware, we have the following

$$b_2 = \left| \left(\left((|B|_{q_2} - b_0) \cdot \left| q_0^{-1} \right|_{q_2} \right) - b_1 \right) \cdot \left| q_1^{-1} \right|_{q_2} \right|_{q_2}. \tag{2.31}$$

If we did not assume that the previous radices are smaller than our current modulus, we would have to compute the modular reduction for the previous radices before subtracting. This is not trivial and limits our flexibility in terms of parameter choices.

The mixed-radix conversion method is useful, where we do not make any assumptions on the input bounds or keep any extra residues; however, they are difficult to parallelize and pipeline on FPGA. Furthermore, their hardware implementations introduce additional limitations in terms of base selection. The bases need to be

Algorithm 4 Garner (1959) Mixed-Radix Conversion Algorithm

Input: $A_Q = a_{q_i}(\forall i)$; $Q = \{q_0, \ldots, q_{l-1}\}$, $P = \{p_0, \ldots, p_{k-1}\}$
Precompute: $\left|q_i^{-1}\right|_{q_h} (\forall i, h | i < h), \left|q_i^{-1}\right|_{p_j} (\forall i, j), |Q|_{p_j} (\forall j)$
Output: $A_P = a_{p_j}(\forall j)$

$b_i \leftarrow a_{q_i}, \theta_{p_j} \leftarrow 0 (\forall i, j)$
for $s = 0, \cdots, l-1$ **do**
  for $u = s+1, \cdots, l-1$ **do**
    $b_u \leftarrow (b_u - b_s) \cdot \left|q_s^{-1}\right|_{q_u} \pmod{q_u}$
  end for
  $\theta_{p_j} \leftarrow (\theta_{p_j} - b_s) \cdot \left|q_s^{-1}\right|_{p_j} \pmod{p_j} (\forall j)$ ▷ *b_s is our mixed-radix digit*
end for
$a_{p_j} \leftarrow |Q|_{p_j} \cdot (-\theta_{p_j}) \pmod{p_j} (\forall j)$

extended in an ordered fashion. This limitation reduces its generalizability for FHE use cases. It assumes access to all residues corresponding to a coefficient in a single iteration, which requires careful organization of the inputs. Finally, for a pipelined implementation, we must compute with $\frac{(l+1) \cdot l}{2}$ words simultaneously, which strains the arithmetic and memory resources of our FPGA.

2.6 Homomorphic encryption libraries

The state-of-the-art software libraries for working with HE are shown in Table 2.1. The list contains the most commonly used libraries, to the best of our knowledge.

Table 2.1 Homomorphic encryption libraries

Note; ✓: supports, ✗: does not support.

HE Library	BFV Support	CKKS Support	Development	Language
OpenFHE	✓	✓	Active	C++
SEAL	✓	✓	Not Active	C++
HELib	✗	✓	Not Active	C++
HEAAN	✗	✓	Active	C++
Lattigo	✓	✓	Active	Go
Swift HE	✓	✗	Active	Swift
TFHE-rs	✗	✗	Active	Rust

Shown in Table 2.1 are the actively developed libraries that have releases within a year. HEAAN library has an older open-source version, but the current version is

closed-source. HEAAN also features GPU acceleration with CUDA.

2.6.1 OpenFHE

OpenFHE (Al Badawi et al. (2022)) is the state-of-the art library with support for BFV, BGV, CKKS, DM, CGGI and LMKCDEY schemes. It has a layered design approach, allowing designers to work on higher or lower abstraction layers.

2.6.2 Microsoft SEAL

Microsoft SEAL (Microsoft (2023)) HE library has support for BFV, BGV, and CKKS schemes. It is a mature library that has been the basis of research works. However, its development has significantly slowed down after 2023. It is being maintained but does not receive significant updates.

2.6.3 Lattigo

Lattigo (Tuneinsight (2024)) library supports BFV, BGV and CKKS schemes. It is particularly targeting distributed and multi-party workloads.

2.6.4 Swift Homomorphic Encryption

Swift Homomorphic Encryption library (Apple Inc. and Swift Homomorphic Encryption project authors (2025)) supports BFV scheme only. This library is the basis of the private information retrieval and private nearest neighbor search applications in Apple devices.

2.6.5 Comparing Against Libraries

Our goal is not to compare the performances of HE libraries but to present our work in the context of software implementations. Comparing and benchmarking HE libraries is not trivial due to their support of different parameters and lower-level implementations. We have chosen OpenFHE as a representative due to its wide range of support for schemes and implementations.

3. AREA-TIME PRODUCT EFFICIENT RNS POLYNOMIAL BASE EXTENSION

This chapter focuses on our work implementing and optimizing base extension architecture on FPGA. We set the context of accelerating arithmetic on FPGAs by providing insight into our design process. We present hardware-friendly algorithms we have utilized and explain why they are advantageous over those found in the literature. Furthermore, we then explain our hardware architecture design for base extension. We introduce a scalable, throughput-oriented, pipelined, and folded architecture for the Shenoy-Kumaresan algorithm. Finally, we compare our architecture with those found in the literature for the FHE use case.

3.1 Loosely Coupled FPGA Accelerator

The end goal of our hardware design is to be integrated in a FPGA loosely coupled accelerator as shown in Figure 3.1. The choice of a loosely coupled accelerator results in a significant communication cost between the host and the device. For our use case, there is a PCIe Gen 3×16 interface between the host and the device, resulting in a simple memory-mapped copy operation between the host device and the FPGA accelerator device taking on the order of milliseconds. The communication overhead of transfers between the host and the device diminishes the practical speedup of the accelerator.

Hence, our goal is to minimize the number of transactions between the host and the device. To achieve this, we implement the necessary algorithms on the device itself. Thus, our base extension algorithm shares the programmable logic area with other computationally expensive operations due to other homomorphic computations such as NTT and relinearization . Our goal is to use as much of the area for actually useful computation. For this reason, we choose ATP as our figure of merit.

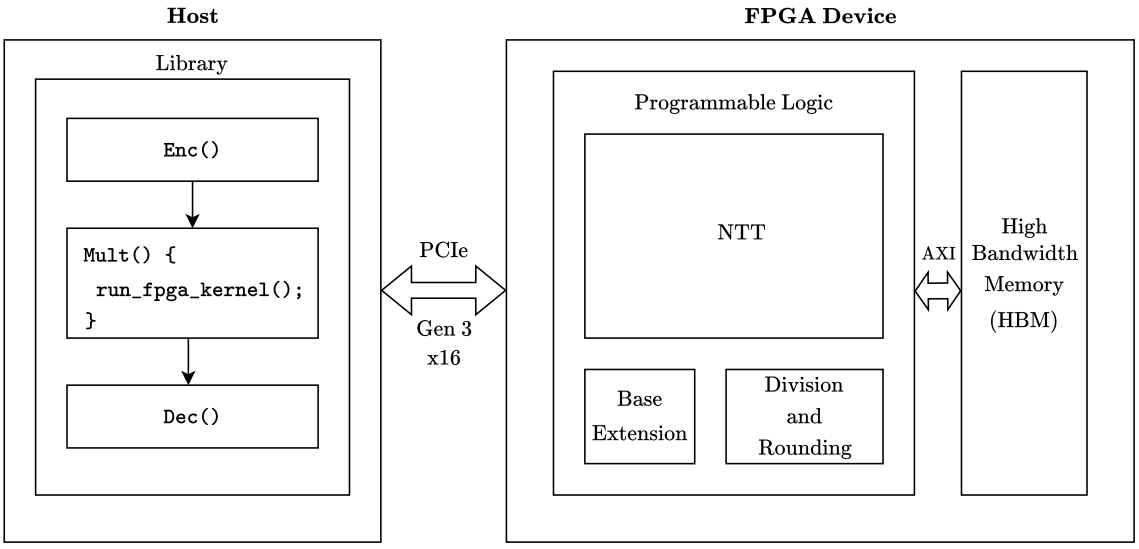


Figure 3.1 High-level overview of the accelerator.

As the number of polynomials required to compute homomorphic multiplication increases with the number of RNS residues, the capacity of memory needed exceeds the total available on-chip memory capacity of 43 MB, including BRAM and URAM (AMD Xilinx (2023)). So, our design also makes use of the off-chip HBM available on the Alveo U280 FPGA, which has a total capacity of 8 GB (AMD Xilinx (2023)).

The Alveo U280 FPGA has 32 AXI-3 channels that are connected to the HBM subsystem as shown in Figure 3.2. There are two memory stacks of 4 GB capacity. The physical connection between the FPGA die and the HBM is facilitated through a silicon interposer. HBM subsystem consists of 8 Mini Switch (MS)s that connect 16 Memory Controller (MC)s. Finally, each MC is connected to two Pseudo Channel (PC)s that each have a theoretical maximum bandwidth of 14.375 GB/s. Each PC is directly connected to a 256 MB section of the HBM. The HBM memory subsystem can run at native 450 MHz or 225 MHz through the usage of BRAMs and First In First Out (FIFO) registers (AMD Xilinx (2024)). Attaining 450 MHz native frequency is difficult for the FPGA programmable logic, so we will assume that we are communicating with the HBM memory subsystem at 225 MHz. At 225 MHz, we are given 16384-bit total bandwidth to and from the HBM. As this is the theoretical maximum, meaning it does not take into account the backpressure mechanism, switching delays, and refresh cycles, we may not be guaranteed this bandwidth at all times. In our design, we have assumed a 2048-bit guaranteed throughput. The reads and writes will be handled through an intermediary control logic, which will buffer the inputs and outputs of our design.

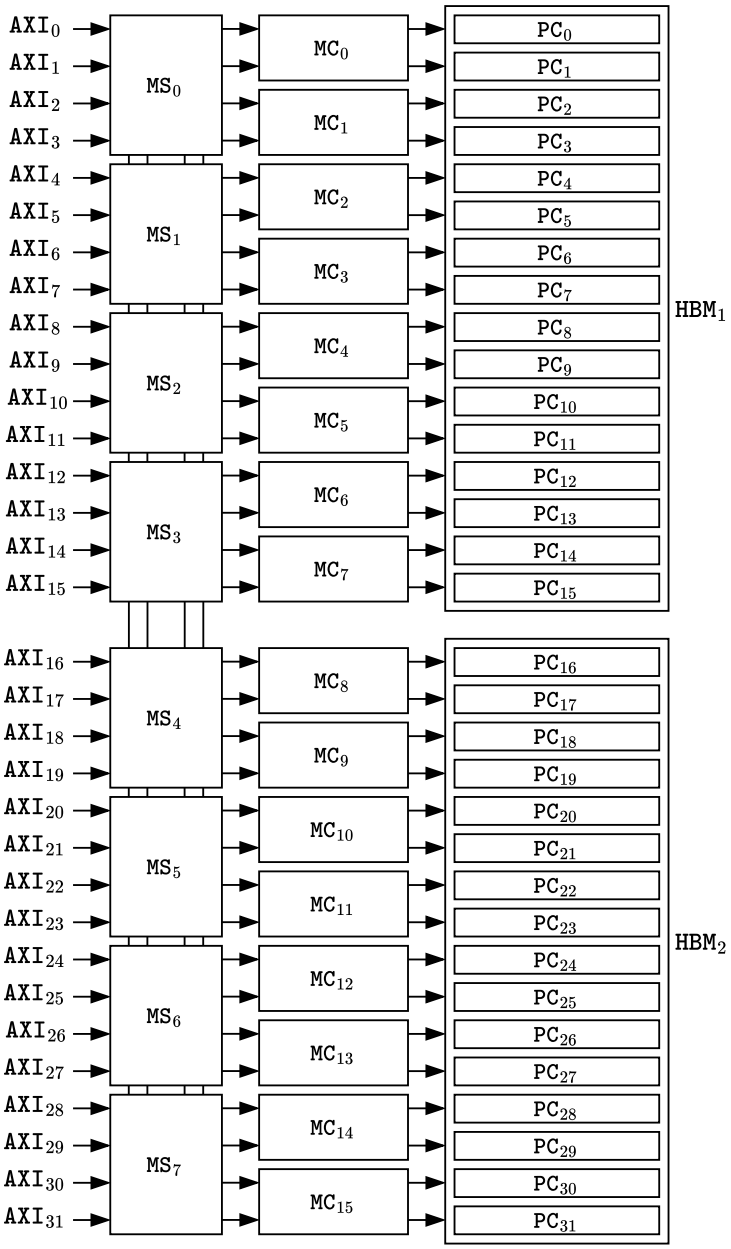


Figure 3.2 HBM interface on AMD Alveo U280 FPGA (AMD Xilinx (2022))

3.2 Efficient Arithmetic on FPGA

Implementing arithmetic on FPGA efficiently requires some understanding of the resources available on the device. FPGAs have distributed Look-Up Table (LUT) for configurable logic and Digital Signal Processing (DSP) slices for faster arithmetic. These DSP blocks implement integer multiply-accumulate functions. However, unlike CPU or GPU, they generally do not have floating-point arithmetic units. There are implementations for floating-point arithmetic, but the device itself does not have native support. Similarly, performing modular arithmetic requires consideration of the capabilities of the FPGA. An example of a fundamental limitation is the lack of division operation by integers apart from powers of two.

3.2.1 Modular Arithmetic

Performing modular arithmetic in a pipelined manner on FPGA requires that we have deterministic runtime algorithms. Modular additions and subtractions are trivial. For example, for additions shown in Algorithm 5, we guarantee the input range to be between $[0, q-1]$. Then, we check if the sum is larger than q and subtract q if so. Similarly, as seen in Algorithm 6, we can check if our subtraction yields a negative number and add q to the result for the hardware implementation of modular subtraction.

Algorithm 5 Modular Addition Algorithm

Input: $a, b \in [0, q), 2^{r-1} \leq q < 2^r$, assuming arithmetic in binary
Output: $c = a + b \pmod{q}$

1: $e \leftarrow a + b$
2: $f \leftarrow a + b - q$
3: **if** $f_{\{r-1\}} = 1$ **then** ▷ *Result is negative*
4: $\quad c \leftarrow e$
5: **else**
6: $\quad c \leftarrow f$
7: **end if**

For modular multiplications, the lack of division means there are a few algorithms available. These algorithms have trade-offs in terms of precomputed values, run-time complexity, and logic usage. We are using the Word-Level Montgomery reduction algorithm (Algorithm 7) from Mert et al. (2020) with optimizations from Tosun et al.

Algorithm 6 Modular Subtraction Algorithm

Input: $a, b \in [0, q), 2^{r-1} \leq q < 2^r$, assuming arithmetic in binary
Output: $c = a - b \pmod{q}$

1: $e \leftarrow a - b$
2: **if** $e_{\{r-1\}} = 1$ **then** ▷ *Result is negative*
3: $\quad c \leftarrow e + q$
4: **else**
5: $\quad c \leftarrow e$
6: **end if**

(2024). The implementation of the modular multiplier, adder, and subtraction units includes pipeline stages. Hence, they are pipelined sequential circuits. This choice enables us to enhance our throughput while meeting the frequency requirements of our application.

Algorithm 7 Word-Level Montgomery Reduction Mert et al. (2020)

Input: $a \in [0, q), 2^{r-1} \leq q < 2^r$, $\omega | q = 1 \pmod{2^\omega}; q_H = q_{[r-1:\omega]}$, $\nu = \lceil \frac{r}{\omega} \rceil$
Output: $d = a2^{-\omega\nu} \pmod{q}$

1: $T \leftarrow a$
2: **for** $s = 0, \ldots, \nu - 1$ **do**
3: $\quad T_L \leftarrow T_{[\omega-1:0]}$, $T_H \leftarrow T_{[r-1:\omega]}$
4: $\quad \tilde{T} \leftarrow -T_L \pmod{2^\omega}$
5: $\quad \text{carry} \leftarrow \tilde{T}[\omega - 1] \bigvee T[\omega - 1]$
6: $\quad T \leftarrow T_H + q_H T_L + \text{carry}$
7: **end for**
8: **if** $T \geq q$ **then**
9: $\quad d \leftarrow T - q$
10: **else**
11: $\quad d \leftarrow T$
12: **end if**

The $2^{-\omega\nu}$ factor introduced by Montgomery reduction is handled through the pre-processing of the data. For example, multiplying the constants by $2^{\omega\nu}$ by the number of multiplications that will be performed.

3.3 Architecture Overview

Accelerating base conversion in this work aims to exploit parallelism. Exploiting parallelism requires that we eliminate potential branching conditions. Conditional

execution and branch prediction are non-trivial tasks and impact algorithm performance. Even occasional bubbles in the execution have to be handled accordingly.

To exploit this parallelism and perform operations in a throughput-oriented manner, we implement pipelined data paths for consuming inputs and producing outputs. Due to the nature of the base extension algorithm (see Algorithm 2), we cannot proceed any further before we compute the summation of all the results. We therefore have to split the computation into two separate stages. These stages themselves are pipelined, but their execution happens serially. We refer to the first stage as "Sum Process" and the second stage as "Post-Sum Process." Steps 1-3 in Algorithm 2 implement the Sum Process, and steps 4-7 correspond to the Post-Sum Process. Notice that the Sum Process is almost identical in nature to Algorithm 1 . The only difference is the data path corresponding to q_l. The Sum Process stage starts as soon as we receive polynomial coefficients and ends after we have accumulated all the summation results.

Although serializing computation by splitting the execution is not generally desired, it opens up potential optimizations for hardware implementation. Our use case requires base extension hardware to be integrated onto the same FPGA as a general-purpose FHE accelerator. Therefore, any additional LUT or DSP usage can cause the entire design to not fit on the device's programmable logic. We address this problem in two ways. First, we present a design-time configurable hardware generator that allocates FPGA resources proportional to the throughput performance requirements. Second, we reuse multipliers for the Sum and Post-Sum Processes. Most of the multipliers on the FPGA are consumed by the NTT accelerator unit, and saving from valuable DSP resources is crucial for fitting the accelerator to the FPGA.

3.3.1 Data Flow of Algorithm 1

In this section, we provide insights into the data flow of Algorithm 2, a crucial step in designing the hardware for the algorithm. Our aim is to show the approach we took in designing the architecture.

Here, we notice that the result of step 1 in Algorithm 2 is the modular multiplication with respect to q_i, which can be used in both α and α_{p_j} computation, as observed in Figure 3.3. Hence, this computation is broadcast, and a single set of modular multiplier hardware for computing μ_{q_i} is enough. Additionally, we would like to note

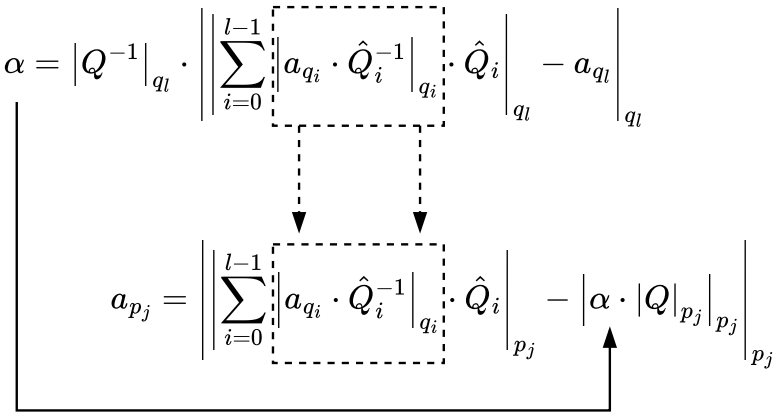


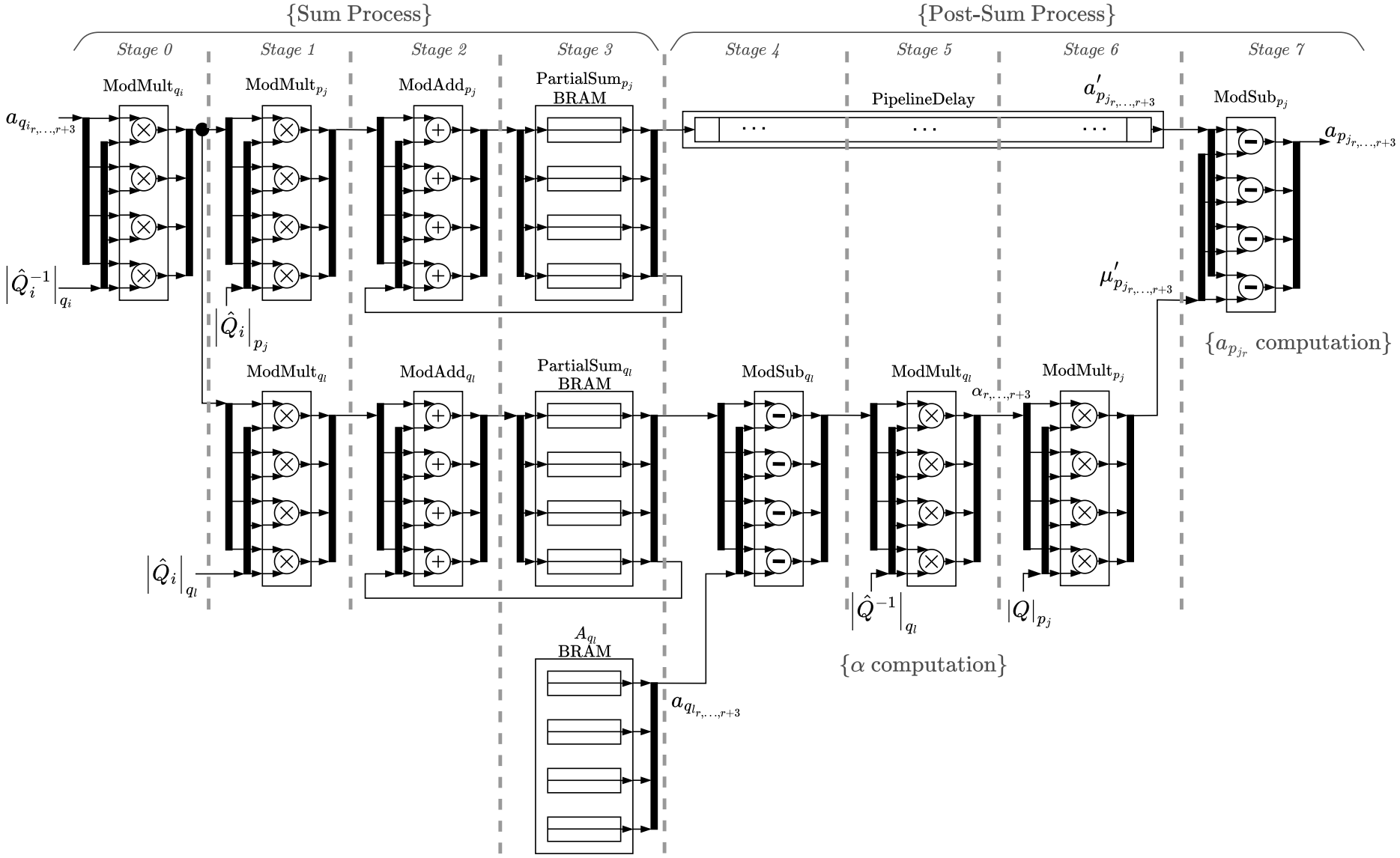
Figure 3.3 Data flow of Shenoy-Kumaresan algorithm, where $j = 0, \ldots, k-1$

that we must first compute α before proceeding with the subsequent computations. Hence, we implement our architecture in accordance with these dependencies. We would also like to distinguish the clock cycle (CC) and iteration count measures. Due to the pipelining stages and internal read/write logic, there may be changes in the CC count. Hence, we will use an iteration count that abstracts away the underlying complexity. Our conclusions will still hold because filling the pipeline stage can take up to ≈ 50 CCs, whereas our computations scale with N, which is orders of magnitude larger.

3.3.2 Fully Pipelined Architecture

In this section, we present our architecture, which implements Algorithm 2 in a fully pipelined manner. Here, the term fully pipelined signifies that it can accept a new set of residues corresponding to a different polynomial when the previous one has been computed for the given stage. Hence, no bubbles would be needed to compute subsequent base extension operations for a different polynomial. The architecture is shown in Figure 3.4. We omit the pipeline registers used to synchronize the computations in the figure for clarity, but divide the computation into the pipeline stages, indicating the synchronization points. Also, note that, as shown in subsection 3.2.1, modular arithmetic modules are also sequential circuits, each with its own pipeline stages. The stage and clock cycle counts of modular arithmetic blocks vary according to the specified parameters. Hence, this complexity is abstracted away in the figures. The figures are drawn given that $TP = 4$.

Figure 3.4 Architecture overview for a fully pipelined version where $TP = 4$



Notice that we are broadcasting the same second operand for all of the multiplications. This allows us to simplify the read control logic and only access one precomputed term per multiplier group. For larger designs, access to memory can be further pipelined to avoid critical paths caused by broadcasting. In our designs with $TP \leq 64$, the critical paths contributed by these broadcast accesses were not large enough to justify the extra design complexity.

Our approach implements the data flow shown in Figure 3.3 with pipeline stages synchronizing the consumption and production of intermediate results. Computing α requires us to instantiate an execution pipeline where the operations are performed modulo q_l. While computing α, we have to delay the a'_{p_j} intermediate results with pipeline registers to ensure that ModSub_{p_j} takes the correct order of inputs.

The control logic of our design consists of simple counters that keep track of the progress of the Sum and Post-Sum Processes. We loop through the Sum Process $\frac{N \cdot (l+1)}{TP}$ times. Please note that the first iteration over $\frac{N}{TP}$ is spent on ingesting the redundant residue to the A_{q_l} BRAM group. Subsequent $\frac{N \cdot l}{TP}$ iterations are used to compute the a'_{q_l} and a'_{p_j} partial sums. We can't progress before we compute these partial sums.

Computing these sums separately for q_l and p_j is a design requirement. We can't work with the full precision $\hat{Q}_i$'s. From subsection 2.5.1 we can see that the bit width of $\hat{Q}_i$ is $\lceil \log_2 (Q - q_i + 1) \rceil$. For a word size of 64 bits and $l = 17$, a single $\hat{Q}_i$ would become 1024 bits. We would have to compute with thousand-bit operands, which would diminish the utility of RNS and pose significant resource usage issues on our FPGA.

A fully pipelined architecture is useful when we frequently need to extend independent polynomials, meaning RNS residues corresponding to distinct polynomials. However, as we will introduce in the next section, we can improve our resource consumption significantly by relaxing the constraint of accepting distinct polynomial residues immediately after the previous one is computed for the Sum Process.

3.3.3 Folded Architecture

In the abstract overview of the architecture shown in Figure 3.5, the modular multipliers for q_l and p_j are used again for the Post-Sum Process. When coupled with an n-step NTT architecture and given that the throughput is not bottlenecked by base extension, we can reuse the same modular multipliers, specifically $2 \cdot TP$ modular

multipliers for both processes. Essentially, this is possible since base extension will not accept a new set of polynomial inputs when it starts computing residues. Note that we are assuming that $N > l$, which is the case in practice.

Figure 3.5 Folded architecture overview where $TP = 4$

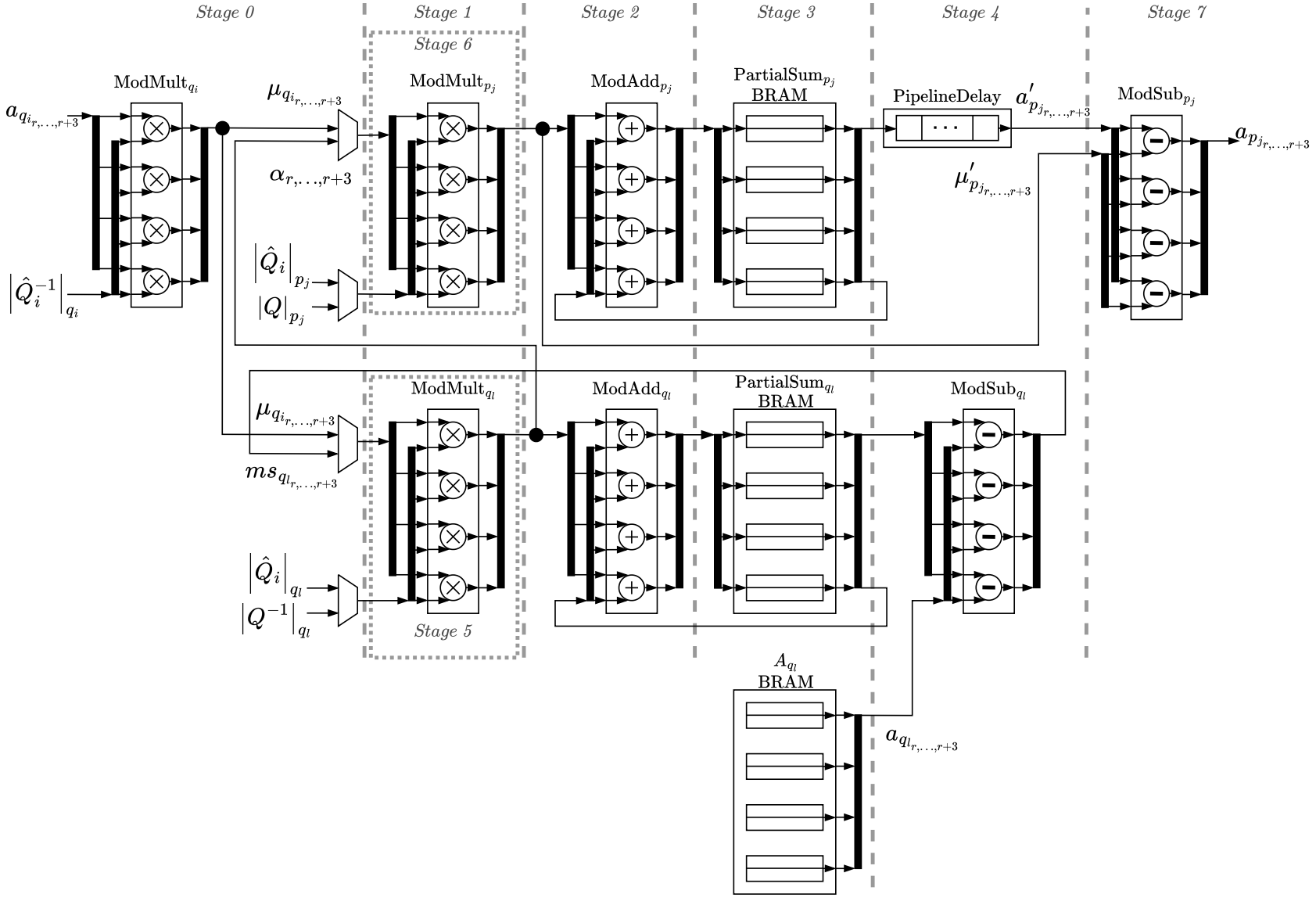
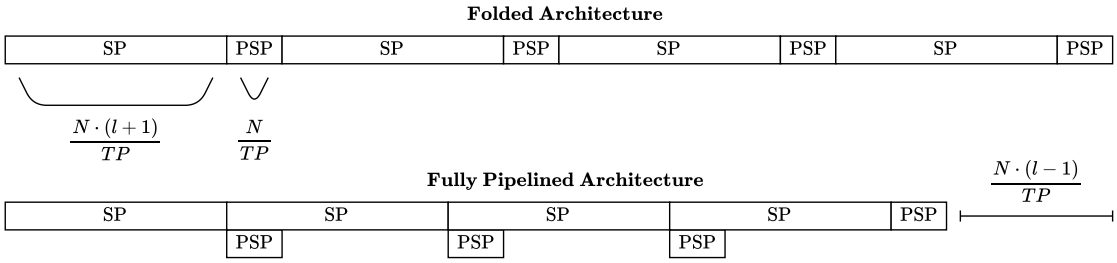


Figure 3.6 Architecture comparison in terms of number of iterations

Note; SP: Sum Process, PSP: Post-Sum Process.



By computing the Sum and Post-Sum Processes serially, we increase the iteration count by $\frac{N \cdot (l-1)}{TP}$ iterations, almost by one length of the Sum Process. We can observe an example execution shown in Figure 3.6. Due to the interface between the FPGA and HBM, shown in Figure 3.2, our bandwidth is bottlenecked by the amount of simultaneous transfers. Hence, a read and write operation consumes from the same total bandwidth available on the device. Thus, in practice, avoiding simultaneous read and write operations is preferable. We observe that, by performing Sum and Post-Sum Processes mutually exclusively in the Folded architecture, we only consume and produce TP-many coefficients as opposed to $2 \cdot TP$ coefficients in the Fully Pipelined architecture. Which is advantageous for our application, where we are guaranteed a limited bandwidth for the HBM interface.

An example case would be with $N = 2^{16}, \quad TP = 64, \quad l = 30$, where the Folded architecture would take 983040 iterations and the Fully Pipeline architecture would take 953344 iterations. The difference is 29696 iterations or $\approx 3.2\%$ of the Fully Pipeline architecture iteration count. The design choice of time-multiplexing the computation can be revisited if the base-extension unit becomes the bottleneck during computation. These stages can be separated again, given that we satisfy the bandwidth limitations, to accept a new input set after completing the sum process.

3.3.3.1 Memory Requirements

Computing Algorithm 2 requires that we store the following precomputed terms shown in Table 3.1. They are all residues with the size $\lceil \log_2 (q_i + 1) \rceil$.

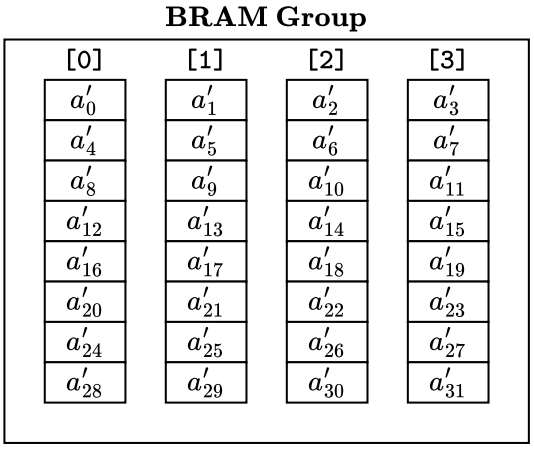
To store the summation results for subsequent computations, we instantiate two BRAMs with depth N. Another BRAM is required to store the redundant residues separately to compute the Post-Sum Process. In total, we instantiate three BRAMs

Table 3.1 Number of precomputed terms required by Shenoy-Kumaresan

Precomputed Term	**# of terms to store**
$\left\|\hat{Q}_i^{-1}\right\|_{q_i}$	l
$\left\|\hat{Q}_i\right\|_{p_j}$	$l \cdot k$
$\left\|\hat{Q}_i\right\|_{q_l}$	l
$\left\|Q^{-1}\right\|_{q_l}$	1
$\left\|Q\right\|_{p_j}$	k

with depth N. Each BRAM stores coefficients with the method shown in Figure 3.7.

Figure 3.7 Example BRAM coefficient placement for $TP = 4$, $N = 2^5$



The BRAM coefficient placement can be altered easily, as the computation only requires that we are consistent across other BRAM groups as well. For example, it is not necessary that we receive coefficients in an ordered fashion. As long as we receive the subsequent coefficients in the same order, the computation will be correct.

3.3.3.2 Arithmetic Requirements

Arithmetic units are considered the area usage required to implement the architecture. Here, we count the modular subtraction units as the same as the modular addition units to simplify analysis, as they are identical in resource consumption. The total arithmetic unit count can be seen in Table 3.2.

Table 3.2 Number of arithmetic units required by Shenoy-Kumaresan to produce one residue

Architecture	Arithmetic Unit	Count
Fully Pipelined	Modular Adder	$4 \cdot TP$
	Modular Multiplier	$5 \cdot TP$
Folded	Modular Adder	$4 \cdot TP$
	Modular Multiplier	$3 \cdot TP$

3.4 Optimizations for Polynomial Residues

Our accelerator is optimized for coupling with a FHE accelerator. The target NTT accelerator in the final implementation processes polynomials for one residue at a time. Hence, NTT takes inputs and outputs of size N polynomial for one residue at a time. Our approach is optimized for this type of computation. We aim to overlap this process with our own computation. Thus, when the NTT accelerator finishes the last residue polynomial, we will be able to output the first residue polynomial in the target base within N/TP iterations.

This method of computation also avoids the costly and complicated reordering of the coefficients by the preceding and subsequent units. By being flexible and not enforcing a coefficient storage strategy, we enable optimizations on algorithms other than base extension. Hence, as an example, NTT units can reorder and store their results in the most optimal way and not have to be burdened by the processing of the base extension unit.

3.4.1 Avoiding Communication Overhead by Integration

For one homomorphic multiplication $3 \cdot N \cdot l$, coefficients are consumed by relinearization. An example case is for the parameter set $\log_2 q_i = 60, N = 2^{16}, l = 18$, the data to be copied becomes 26.5 MiB. Ignoring all other overheads, sending this data would take 1.65 ms over a PCIe Gen 3×16 connection to the host. We will then copy this data back to the device for the remaining operations, which effectively results in at least 3.3 ms of communication overhead, even excluding any computation costs on the host side.

We avoid this overhead by staying on the device and avoiding transfers with the

host. This also frees up the host to perform other useful tasks, such as preparing for the next offloading of the computation. Furthermore, we also save on the energy of these expensive data movements to and from the host.

4. RESULTS AND COMPARISON

Here, we present our results and comparison with the available architectures for base extension. We then provide our utilization results on FPGA and finally present our work in the context of the software implementation.

4.1 Architecture Comparison

In this section, we will compare our approach to the ones available in the literature. Our goal is to show the justifications for the design choices being made in this work. We will not include Algorithm 1 in this comparison as it is already part of the Algorithm 2 as the "Sum Process".

4.1.1 Cox-Rower Architecture

Cox-Rower architecture is specialized for use in the RSA public key cryptography algorithm. It implements Algorithm 3 with units specifically tailored for the parameters given. It exploits the symmetry of the base extension operation in RNS Montgomery multiplication. The source base and the target base both are comprised of l-many residues. There are subsequent works (Nozaki et al. (2001)) introducing methods to time-multiplex this architecture, but they require handling edge cases where the number of processing elements has to be a factor of the number of residues to avoid overheads. We will primarily focus on the original work by Kawamura et al. (2000), which simplifies our analysis. We assume that our concerns regarding the compatibility with NTT and limited input range are solved for this part of the analysis.

4.1.1.1 Memory Requirements

We have to store the terms shown in Table 4.1 in the read-only memory to compute Algorithm 3 in the Cox-Rower architecture.

Table 4.1 Number of precomputed terms to store by Cox-Rower architecture

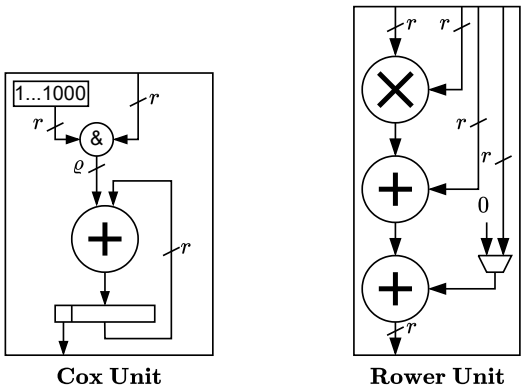
Precomputed Term	**# of terms to store**
$\left\|\hat{Q}_i^{-1}\right\|_{q_i}$	l
$\left\|\hat{Q}_i\right\|_{p_j}$	l^2
$\left\|-Q\right\|_{p_j}$	l

Another requirement of the Cox-Rower architecture is the availability of the residues that represent an integer in the source base simultaneously. When computing Line 4 of Algorithm 3 we assume that we are able to access all of the source residues in a single clock cycle. This may introduce complexity in terms of design considerations. We will refer to this point in our comparison with our work to further discuss it.

4.1.1.2 Arithmetic Requirements

The architecture consists of two arithmetic units, Cox and Rower, that are shown in Figure 4.1. The higher-level architecture can be seen in Figure 4.2. Cox unit computes Lines 1-3,7,8, and 9 in Algorithm 3. It has a ϱ-bit adder and a register that produces the bits of α.

Figure 4.1 Cox and Rower units



The Rower units have a multiplier, an accumulator, and a modular reduction unit that computes Line 10. The multiplication with ξ_{q_s} has a bit width of r. Multiplica-

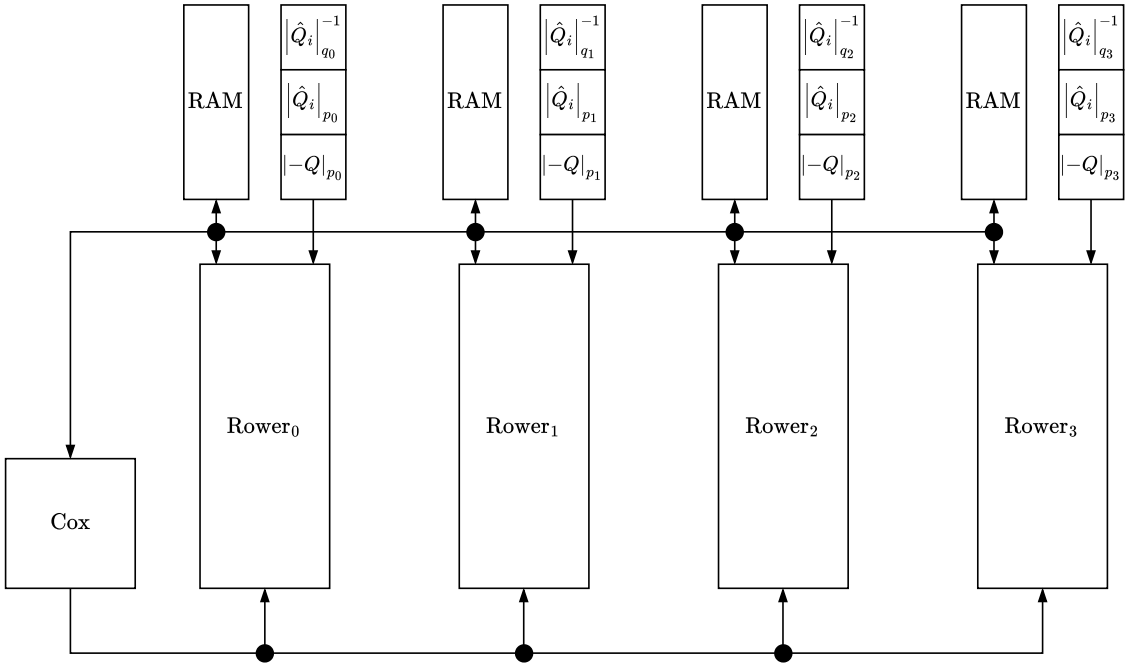


Figure 4.2 Cox-Rower architecture overview

tion with the $|-Q|_{p_j}$ term is with a single bit $\alpha_{\{s\}}$, which can be implemented with a multiplexer instead. The authors designed this datapath to be implemented within a Very-large-scale Integration (VLSI) ASIC chip. Thus, the combinational critical path timing issues with FPGA implementations were not considered. If we were to adapt this design to an FPGA, we would have to use pipelined adders and modular multipliers instead. We will take this modification into account when comparing the circuit complexity between the architectures. Thus, a single rower unit, for our analysis, consists of two modular adders and one modular multiplier, as shown in Table 4.2.

Table 4.2 Number of arithmetic units required by Cox-Rower architecture to produce l residues

Arithmetic Unit	**Count**
Adder	1
Modular Adder	$2 \cdot l$
Modular Multiplier	l

4.1.1.3 Comparison to Our Work

Given that we are able to find suitable primes and tolerate the reduction in the representation range, Cox-Rower architecture computes $N \cdot l$ target base residues in

$N \cdot (l+1)$ iterations. This approach directly couples the circuit complexity with the number of residues in the source and target bases. It is less flexible and configurable compared to our decoupled throughput compile-time design parameter. Adapting this approach to be flexible requires handling non-trivial design decisions, such as incorporating an address generation unit or a more complex finite-state machine.

4.1.2 Mixed-Radix Architecture

Algorithm 4 is utilized in converting our RNS non-positional representation to a positional representation. It provides a unique ability to determine the order between two integers that were in the RNS representation. This ability is necessary to compute algorithms, where comparisons are required to determine the execution path. We will assume a fully parallel implementation due to the nature of the Mixed-Radix algorithm. It has been utilized in massively parallel systems, such as GPUs, as shown in Al Badawi et al. (2021). Algorithm 4 can be implemented in a variety of configurations. In Figure 4.3 we present one such architecture.

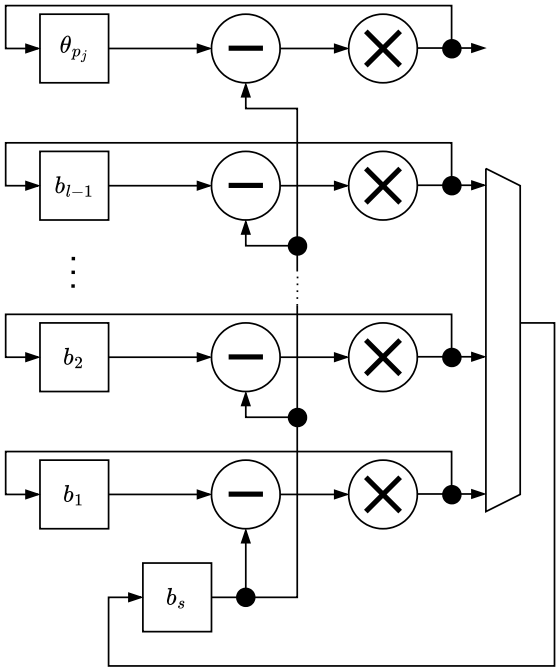
4.1.2.1 Architecture Design

In this section, we explain the reason we have taken the iterative design, as shown in Figure 4.3, into account.

In Figure 4.3, we present one design that computes Algorithm 4 iteratively. It is unrolling the inner for loop with respect to u, computing steps 3-6. It is an abstract view of a design that computes a single new residue in $l+1$ iterations with l modular multipliers.

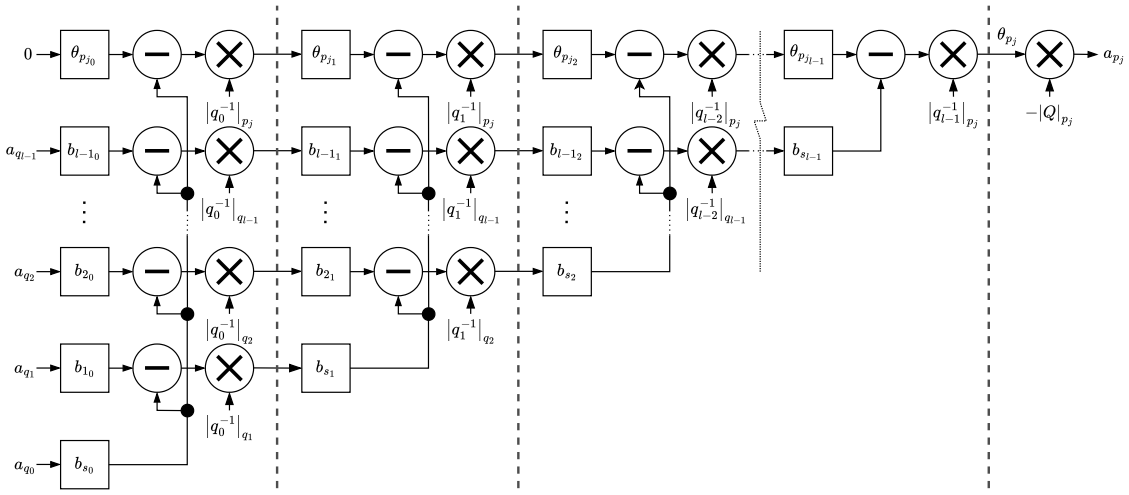
Conversely, we could have implemented Algorithm 4 with unrolling the outer loop as well. We will assume that we are producing a single new residue. Utilizing $\frac{(l+1)\cdot l}{2} + 1$ modular multipliers, we could compute a new residue in every single iteration. However, this would quickly exhaust the DSP blocks on our FPGA. For $l = 30$ this would mean that we would instantiate 466 modular multipliers. If we assume a 32-bit word size and 6 DSPs for each modular multiplier, we would use 2796 DSP blocks, which is almost a third of the 9024 total available DSP blocks on our Alveo U280 FPGA (AMD Xilinx (2023)). With an increasing number of residues,

Figure 4.3 Mixed-Radix iterative architecture overview



the multiplier usage will increase quadratically, which exacerbates the problem of high resource usage.

Figure 4.4 Mixed-Radix fully pipelined architecture overview



Hence, we have chosen the iterative design for our comparisons. The higher-level view of the architecture is shown in Figure 4.3. This implementation instantiates as few parallel multipliers as possible to compute Algorithm 4.

4.1.2.2 Memory Requirements

Precomputed terms for the mixed-radix algorithm are shown in Table 4.3.

Table 4.3 Number of precomputed terms required by Mixed-Radix architecture

Precomputed Term	**# of terms to store**
$\left\|q_i^{-1}\right\|_{q_h}$	$l \cdot (l-1)/2$
$\left\|q_i^{-1}\right\|_{p_j}$	$l \cdot k$
$\|Q\|_{p_j}$	k

Along with these precomputed values, we have to broadcast b_s for the subtraction operation in all residues. This data movement may complicate critical path requirements for FPGA in applications with a large number of residues. Like with Algorithm 3 this algorithm also assumes that we can access all residues that represent an integer simultaneously. Thus, the memory access and bandwidth requirements are again directly coupled to its performance. Lastly, the next b_s has to be determined by a multiplexer that selects the current radix that will be used for the subtraction in the next iteration. This multiplexer will have $l-1$ inputs and one output, with a high number of residues and bit-width, which can increase the complexity of the circuit.

4.1.2.3 Arithmetic Requirements

Computation of subsequent radices cannot continue until we process all updates on the individual residues. The sequential nature of the mixed-radix algorithm requires that we have $l+1$ multipliers and l subtraction units in parallel to compute the updates as shown in Table 4.4. At each iteration, we can retire one residue multiplier as it won't be required for the computation of subsequent residues.

Table 4.4 Number of arithmetic units required by Mixed-Radix architecture to produce one residue

Arithmetic Unit	**Count**
Modular Adder	l
Modular Multiplier	l

4.1.2.4 Comparison to Our Work

Mixed-radix conversion requires that we compute a mixed-radix digit b_s before continuing the computation. Hence, it is less parallelizable. Algorithm 4 can compute new residues in $N \cdot (l+1)$ clock cycles. We have to store all residues corresponding to each polynomial coefficient such that they can be accessed simultaneously. Introducing a throughput design parameter is thus not possible. Implementing fewer than l multipliers would also complicate the design datapath, requiring us to time-multiplex the computation.

Utilizing the proposed folded architecture, we can complete the Sum Process in $\frac{N \cdot (l+1)}{TP}$ and the Post-Sum Process in $\frac{N}{TP}$ clock cycles. Our approach is specialized for extending polynomial residues.

4.1.3 Overview of the Comparison of Algorithms and Architectures

In this section, we present the algorithms and their architectures in a comparable parameter set. Since the most constrained parameter set belongs to the Cox-Rower architecture implementing Algorithm 3 we will use its $k = l$ assumption for a fair comparison. In Table 4.5, we list the algorithms and architectures discussed in this work.

Table 4.5 Comparison of algorithms and architectures where $k = l$

Note; Alg.: Algorithm, Arch.: Architecture, Flex.: Flexible, SK: Shenoy-Kumaresan, RBE: Recursive Base Extension, MXR: Mixed-Radix, Pipe.: Pipelined, Fold.: Folded, CR: Cox-Rower, Iter.:Iterative, MM: Modular Multiplier, MA: Modular Adder, A: Adder, ✓: is flexible, ✗: not flexible,

Alg.	Arch.	Memory	Arithmetic	Iterations	Flex.
SK	Pipe.	$l^2+3 \cdot l+1$	$5 \cdot TP \cdot MM+4 \cdot TP \cdot MA$	$\frac{N \cdot (l+1) \cdot l}{TP}+\frac{N}{TP}$	✓
	Fold.	$l^2+3 \cdot l+1$	$3 \cdot TP \cdot MM+4 \cdot TP \cdot MA$	$l \cdot (\frac{N \cdot (l+1)}{TP}+\frac{N}{TP})$	✓
RBE	CR	$l^2+2 \cdot l$	$l \cdot MM+2 \cdot MA+A$	$N \cdot (l+1)$	✗
MXR	Pipe.	$l^2+\frac{l \cdot (l-1)}{2}+l$	$(\frac{(l+1) \cdot l}{2}+1) \cdot MM+\frac{(l+1) \cdot l}{2} \cdot MA$	N	✗
	Iter.	$l^2+\frac{l \cdot (l-1)}{2}+l$	$l \cdot MM+l \cdot MA$	$N \cdot (l+1)$	✗

We notice a relationship emerging from the number of iterations and arithmetic requirements. As expected, the iteration count decreases as we instantiate more arithmetic units. One such example is the N iteration count, the smallest iteration

count, of the pipelined architecture corresponding to the mixed-radix algorithm, which instantiates $(l+1) \cdot l$ modular multipliers. The Recursive Base Extension algorithm is represented by the Cox-Rower architecture, which instantiates l modular multipliers and performs the base extension in $N \cdot (l+1)$ iterations. Finally, our folded architecture instantiates $3 \cdot TP$ modular multipliers and performs the base extension in $\frac{N \cdot (l+2) \cdot l}{TP}$ iterations.

If we were to define a parameter as the product of modular multipliers and number of iterations, we notice that they are in the form $a \cdot N \cdot l \cdot (l+b)$. Where a and b are tunable according to the choice of algorithm and architecture.

Our architecture is also the only approach where the number of modular multipliers is a design parameter. Both Cox-Rower and mixed-radix architectures require us to instantiate modular multipliers according to the number of residues in the source (l) and target ($k = l$, for this comparison) bases. This flexibility can also enable the use of a single hardware for multiple parameter sets. By specifying a maximum l, the counter logic could be modified to accommodate multiple parameter sets in the future. Other architectures are not flexible in this regard. Accommodating variable parameter sets would require significant design effort for the control logic, if even possible.

Table 4.6 Comparison of iteration counts for given parameters

Algorithm	**Architecture**	**Iterations**		
		$N = 2^{16}$	$N = 2^{14}$	$N = 2^{12}$
		$TP = 32$	$TP = 32$	$TP = 64$
		$l = 17$	$l = 7$	$l = 4$
Shenoy-Kumaresan	Pipelined	628736	29184	1344
	Folded	661504	32256	1536
Recursive Base Extension	Cox-Rower	1179648	131072	20480
Mixed-Radix	Pipelined	65536	16384	4096
	Iterative	1179648	131072	20480

Although our architecture is more complex in theory, it performs better in practical parameter sets as shown in Table 4.6. The advantage of our architecture increases when TP is larger than l. This behavior is expected as if we plug $TP = l$, our Folded architecture iteration count becomes $N \cdot (l+2)$.

4.2 Utilization Results

In this section, we present our utilization results for the given parameters. We have implemented the designs on AMD Alveo U280 using Vivado 2022.2 with SystemVerilog HDL. Here, the given parameters are chosen from modifying examples in OpenFHE for homomorphic multiplication with the BEHZ method. We compute the ATP metric as follows: $\text{Time} \cdot (\text{LUT} + \frac{\text{FF}}{2} + 100 \cdot \text{DSP} + 300 \cdot \text{BRAM})$.

Table 4.7 Device utilization results for $\log N = 16, l = 17$

$\log q_i$	TP	LUT ($\times 10^3$)	FF ($\times 10^3$)	BRAM	DSP	t_{min} (ns)	f_{max} (MHz)	CC	Time (μs)	ATP ($\times 10^6$)
32	1	2.4	5	192	18	2.1	476.19	1245184	2614.89	168.18
	16	22.5	37.1	192	288	2.1	476.19	77824	163.43	20.82
	64	75.8	139.9	192	1152	2.7	370.37	19456	52.53	16.73
64	1	5.9	12.9	384	60	2.55	392.16	1245184	3175.22	424.03
	8	31.5	53.9	384	480	2.7	370.37	155648	420.25	93.16
	32	121.1	198.5	384	1920	3.2	312.50	38912	124.52	65.70

Table 4.8 Device utilization results for $\log N = 14, l = 7$

$\log q_i$	TP	LUT ($\times 10^3$)	FF ($\times 10^3$)	BRAM	DSP	t_{min} (ns)	f_{max} (MHz)	CC	Time (μs)	ATP ($\times 10^6$)
32	1	1.8	3.6	48	15	2	500.00	147456	294.91	5.75
	16	20.7	35.5	48	240	2.1	476.19	9216	19.35	1.49
	64	70	136.9	96	960	2.7	370.37	2304	6.22	1.64
64	1	5.4	10.8	96	66	2.45	408.16	147456	361.27	16.68
	8	32.6	54.9	96	528	2.55	392.16	18432	47.00	6.66
	32	124.8	206	96	2112	3.3	303.03	4608	15.21	7.11

Table 4.9 Device utilization results for $\log N = 12, l = 30$

$\log q_i$	TP	LUT ($\times 10^3$)	FF ($\times 10^3$)	BRAM	DSP	t_{min} (ns)	f_{max} (MHz)	CC	Time (μs)	ATP ($\times 10^6$)
32	1	3.0	6.7	12	21	1.95	512.82	131072	255.59	3.09
	16	24.2	41.7	24	336	2.1	476.19	8192	17.20	1.48
	64	80.7	153.8	96	1344	2.77	361.01	2048	5.67	1.82
64	1	7.4	16.6	22.5	66	2.4	416.67	131072	314.57	9.13
	8	34.9	61.3	24	528	2.55	392.16	16384	41.78	5.25
	32	128.1	215.5	96	2112	3.4	294.12	4096	13.93	6.63

We provide results for up to 2048-bit throughput, consistent with our assumption in Chapter 3. We can notice that our DSP usage is different for each polynomial size. This is due to the optimization provided by Tosun et al. (2024). Where the modulus has fewer free significant bits for reduction in higher ring dimensions. We also note that our ATP metric improves as we increase parallelism, which substantiates our

scalability claim. Table 4.7 has the most significant BRAM usage as it has the largest polynomial degree.

Our goal in providing the f_{max} figure is not to implement the designs in the given frequencies but to demonstrate that the critical path will not be from this module when integrated with a complete HE accelerator design. Synthesizing all our designs, for example, in 200 MHz would also strengthen our scalability argument due to the ATP metric.

We also note that the time shown is for computing a single new residue in the target base and should be multiplied by the number of new residues for a complete operation.

Our designs are also synthesizing within the expected frequencies. Increasing the polynomial size or the number of residues does not significantly impact the critical path.

4.3 Comparison with OpenFHE

Here we present the timing results, shown in Table 4.10, Table 4.11, of the Shenoy-Kumaresan base extension implementation in the OpenFHE library. We also include our expected timing results, assuming a clock period of 5 ns. The parameter sets were generated using OpenFHE. Our design supports the given parameters as long as suitable primes are available for the given parameter set. Specifically, our modular reduction method has been tested up to $N = 2^{16}$ with $\log q_i = 32$ and 64.

Table 4.10 Timings for $\log N = 16, \log q_i = 60$

Platform	**Implementation** / l	**17**	**18**
AMD R9 7950X	OpenFHE w/ OMP	15328	16067
	OpenFHE w/o OMP	93856	109454
M1 Pro 10C	OpenFHE w/ OMP	13384	14553
	OpenFHE w/o OMP	48135	54718
AMD TR Pro 3955WX	OpenFHE w/ OMP	19446	21629
	OpenFHE w/o OMP	53087	58512
Intel i7-9750H	OpenFHE w/ OMP	21887	22754
	OpenFHE w/o OMP	55968	65741
Alveo U280	Our work (TP=32)	826.88	921.6

As shown in Table 4.10 Our implementation with Folded architecture achieves $\times 16$

speedup over the best result for $l = 17$. Similarly, we achieve a $\times 15.7$ speedup over the best result for $l = 18$.

Table 4.11 Timings for $\log N = 14, \log q_i = 60$

Platform	**Implementation** / l	**4**	**5**	**6**	**7**
AMD R9 7950X	OpenFHE w/ OMP	744	848	967	1141
	OpenFHE w/o OMP	1541	2102	2561	3540
M1 Pro 10C	OpenFHE w/ OMP	835	968	1093	1234
	OpenFHE w/o OMP	1748	2130	2649	3415
AMD TR Pro 3955WX	OpenFHE w/ OMP	832	942	1086	1282
	OpenFHE w/o OMP	1678	2176	2624	3328
Intel i7-9750H	OpenFHE w/ OMP	1153	1368	1532	1743
	OpenFHE w/o OMP	1678	2176	2624	3328
Alveo U280	Our work (TP=32)	61.44	89.6	122.88	161.28

For $N = 2^{14}$, shown in Table 4.11, our Folded architecture achieves $\times 14$ speedup over the software implementation for $l = 4$. Our speedup becomes $\times 7$ for $l = 7$ compared to the best result.

We have measured the timings with (w/ OMP) and without (w/o OMP) the parallelization provided by OpenMP. Hence, we can observe that although parallelism does speed up the computation, the increase in the number of cores does not directly translate to increased performance. Our understanding is that the communication bandwidth between the CPU dies and the memory is limiting the speedup from increased parallelism. Another performance consideration is the overhead of synchronization primitives required by OpenMP to implement the parallelism. Increasing the number of threads increases the scheduling workload on the CPU.

We notice that as the number of residues and polynomial size increase, our speedup also increases. A possible explanation for why some devices with a high number of cores do not perform better than those with fewer cores is the bandwidth between the CPU and memory. Hence, our architecture with dedicated memory accesses is expected to perform better than even larger, more powerful CPUs due to the memory bottleneck experienced by CPUs.

Timings were measured on four computers. Here are their specifications:

- AMD Ryzen 9 7950X, 16 cores, 4.5 GHz base, 5.7 GHz turbo clock speed, 128 GB of RAM.
- M1 Pro 10 cores, 3.2 GHz $\times$ 8, 2.06 GHz $\times$ 2, 16 GB of RAM.
- AMD Threadripper Pro 3955WX, 16 cores, 3.9 GHz base, 4.3 GHz turbo clock speed, 512 GB of RAM.

- Intel i7-9750H, 6 cores, 2.6 GHz base, 4.5 GHz turbo clock speed, 32 GB of RAM.

5. CONCLUSION AND FUTURE WORK

In this section, we will provide insights resulting from our work and recommend future research avenues for exploration.

5.1 Conclusion

In this work, we have presented a ATP efficient, compile-time configurable, and scalable base extension implementation for accelerating homomorphic multiplication and relinearization operations. Our work explores various base extension methods and compares them against our method. We have demonstrated that our design decouples performance parameters from the underlying arithmetic, and its performance is minimally affected by changes in polynomial size and the number of residues. As our design is parametric, if a better FPGA with higher memory bandwidth becomes available in the future, we would be able to increase our performance simply by increasing the TP parameter. We have shown that we can theoretically achieve up to an order of magnitude speedup compared to the state-of-the-art OpenFHE software library and that our approach is flexible and scalable in its FPGA implementation.

5.2 Future Work

Exploring efficient hardware implementations of algorithms shown in Figure 2.2 is a possible research direction. Unifying the resource consumption of all of the different hardware modules will be necessary for practical implementations on FPGA. One such solution would be to utilize the butterfly units available in the NTT units to

compute the base extensions as well, saving on programmable logic consumption. We also note that our approach is not unique to FPGA applications but can be explored for ASIC designs.

BIBLIOGRAPHY

Agrawal, R., de Castro, L., Yang, G., Juvekar, C., Yazicigil, R., Chandrakasan, A., Vaikuntanathan, V., and Joshi, A. (2023). FAB: An FPGA-based Accelerator for Bootstrappable Fully Homomorphic Encryption. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 882–895.

Al Badawi, A., Bates, J., Bergamaschi, F., Cousins, D. B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., Lee, Y., Liu, Z., Micciancio, D., Quah, I., Polyakov, Y., R.V., S., Rohloff, K., Saylor, J., Suponitsky, D., Triplett, M., Vaikuntanathan, V., and Zucca, V. (2022). OpenFHE: Open-Source Fully Homomorphic Encryption Library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 53–63, Los Angeles CA USA. ACM.

Al Badawi, A., Polyakov, Y., Aung, K. M. M., Veeravalli, B., and Rohloff, K. (2021). Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme. *IEEE Transactions on Emerging Topics in Computing*, 9(2):941–956.

AMD Xilinx (2022). HBM Configuration and Use • Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393) • Reader • AMD Technical Information Portal. https://docs.amd.com/r/2022.2-English/ug1393-vitis-application-acceleration/HBM-Configuration-and-Use.

AMD Xilinx (2023). Summary • Alveo U280 Data Center Accelerator Card Data Sheet (DS963) • Reader • AMD Technical Information Portal. https://docs.amd.com/r/en-US/ds963-u280/Summary.

AMD Xilinx (2024). Introduction • AXI High Bandwidth Memory Controller LogiCORE IP Product Guide (PG276) • Reader • AMD Technical Information Portal. https://docs.amd.com/r/en-US/pg276-axi-hbm/Introduction.

Apple Inc. and Swift Homomorphic Encryption project authors (2025). Swift Homomorphic Encryption. https://github.com/apple/swift-homomorphic-encryption.

Badawi, A. A., Veeravalli, B., Mun, C. F., and Aung, K. M. M. (2018). High-Performance FV Somewhat Homomorphic Encryption on GPUs: An Implementation using CUDA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 70–95.

Bajard, J.-C., Eynard, J., Hasan, M. A., and Zucca, V. (2017). A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes. In Avanzi, R. and Heys, H., editors, *Selected Areas in Cryptography – SAC 2016*, volume 10532, pages 423–442. Springer International Publishing, Cham.

Bajard, J. C., Eynard, J., Martins, P., Sousa, L., and Zucca, V. (2019). Note on the noise growth of the RNS variants of the BFV scheme. https://eprint.iacr.org/2019/1266.

Bossuat, J.-P., Cammarota, R., Chillotti, I., Curtis, B., Dai, W., Gong, H., Hales, E., Kim, D., Kumara, B., Lee, C., Lu, X., Maple, C., Pedrouzo-Ulloa, A., Player, R., Polyakov, Y., Lopez, L., Song, Y., and Yhee, D. (2025). Security Guidelines for Implementing Homomorphic Encryption. *IACR Communications in Cryptology*, 1(4):cc1–4–51.

Brakerski, Z. (2012). Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. https://eprint.iacr.org/2012/078.

Cheon, J. H., Kim, A., Kim, M., and Song, Y. (2017). Homomorphic Encryption for Arithmetic of Approximate Numbers. In Takagi, T. and Peyrin, T., editors, *Advances in Cryptology – ASIACRYPT 2017*, volume 10624, pages 409–437. Springer International Publishing, Cham.

Defense Advanced Research Projects Agency (2021). DPRIVE: Data Protection in Virtual Environments | DARPA. https://www.darpa.mil/research/programs/data-protection-in-virtual-environments.

Fan, J. and Vercauteren, F. (2012). Somewhat Practical Fully Homomorphic Encryption. https://eprint.iacr.org/2012/144.

Garner, H. L. (1959). The residue number system. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference on XX - IRE-AIEE-ACM '59 (Western)*, pages 146–153, San Francisco, California. ACM Press.

Gentry, C. (2009). Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA. Association for Computing Machinery.

Halevi, S., Polyakov, Y., and Shoup, V. (2018). An Improved RNS Variant of the BFV Homomorphic Encryption Scheme. https://eprint.iacr.org/2018/117.

Kawamura, S., Koike, M., Sano, F., and Shimbo, A. (2000). Cox-Rower Architecture for Fast Parallel Montgomery Multiplication. In Goos, G., Hartmanis, J., Van Leeuwen, J., and Preneel, B., editors, *Advances in Cryptology — EUROCRYPT 2000*, volume 1807, pages 523–538. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kim, A., Polyakov, Y., and Zucca, V. (2021). Revisiting Homomorphic Encryption Schemes for Finite Fields. https://eprint.iacr.org/2021/204.

Mert, A. C. (2021). *EFFICIENT HARDWARE IMPLEMENTATIONS FOR LATTICE-BASED CRYPTOGRAPHY PRIMITIVES*. PhD thesis, Sabancı University.

Mert, A. C., Aikata, Kwon, S., Shin, Y., Yoo, D., Lee, Y., and Roy, S. S. (2022). Medha: Microcoded Hardware Accelerator for computing on Encrypted Data.

Mert, A. C., Öztürk, E., and Savaş, E. (2020). Design and Implementation of Encryption/Decryption Architectures for BFV Homomorphic Encryption Scheme. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(2):353–362.

Microsoft (2023). Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

Nozaki, H., Motoyama, M., Shimbo, A., and Kawamura, S. (2001). Implementation of RSA Algorithm Based on RNS Montgomery Multiplication. In Goos, G., Hartmanis, J., Van Leeuwen, J., Koç, Ç. K., Naccache, D., and Paar, C., editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, volume 2162, pages 364–376. Springer Berlin Heidelberg, Berlin, Heidelberg.

Özcan, A. Ş., Ayduman, C., Türkoğlu, E. R., and Savaş, E. (2023). Homomorphic Encryption on GPU. *IEEE Access*, 11:84168–84186.

Rivest, R. L., Adleman, L., and Dertouzos, M. L. (1978). ON DATA BANKS AND PRIVACY HOMOMORPHISMS. *Foundations of secure computation*, 4(11):169–180.

Samardzic, N., Feldmann, A., Krastev, A., Devadas, S., Dreslinski, R., Peikert, C., and Sanchez, D. (2021). F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 238–252, Virtual Event Greece. ACM.

Shenoy, A. and Kumaresan, R. (1989). Fast base extension using a redundant modulus in RNS. *IEEE Transactions on Computers*, 38(2):292–297.

Sinha Roy, S., Turan, F., Jarvinen, K., Vercauteren, F., and Verbauwhede, I. (2019). FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 387–398.

Su, Y., Yang, B.-L., Yang, C., and Zhao, S.-Y. (2022). ReMCA: A Reconfigurable Multi-Core Architecture for Full RNS Variant of BFV Homomorphic Evaluation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(7):2857–2870.

Tosun, T., Kırbıyık, S., Koçer, E., and Alaybeyoğlu, E. (2024). Optimized FPGA Architecture for Modular Reduction in NTT. https://eprint.iacr.org/2024/1890.

Tuneinsight (2024). Lattigo v6. Online: https://github.com/tuneinsight/lattigo. EPFL-LDS, Tune Insight SA.

Turan, F., Roy, S. S., and Verbauwhede, I. (2020). HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA. *IEEE Transactions on Computers*, 69(8):1185–1196.

Van Beirendonck, M., D'Anvers, J.-P., Turan, F., and Verbauwhede, I. (2023). FPT: A Fixed-Point Accelerator for Torus Fully Homomorphic Encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 741–755, Copenhagen Denmark. ACM.