# IMPLEMENTATION OF HIGH SPEED LARGE INTEGER MULTIPLICATION ALGORITHM ON CONTEMPORARY ARCHITECTURE

CHANG BOON CHIAO

MASTER OF ENGINEERING SCIENCE

LEE KONG CHIAN FACULTY OF ENGINEERING AND SCIENCE

UNIVERSITI TUNKU ABDUL RAHMAN

OCTOBER 2018

# IMPLEMENTATION OF HIGH SPEED LARGE INTEGER MULTIPLICATION ALGORITHM ON CONTEMPORARY ARCHITECTURE

By

**CHANG BOON CHIAO**

A dissertation submitted to the Department of Mechatronics and Bio-Medical Engineering,
Lee Kong Chian Faculty of Engineering and Science,
Universiti Tunku Abdul Rahman,
in partial fulfillment of the requirements for the degree of
Master of Engineering Science
October 2018

# ABSTRACT

Cryptosystem plays an important role in cyber security and users privacy protection. To achieve certain level of security, Public Key Cryptography algorithm is performed on large integer that is more than 64-bit, typical bit size supported by conventional Central Processing Unit (CPU) (e.g. 512-bit for Elliptic Curve Cryptography (ECC), 2048-bit for Rivest-Shamir-Adleman (RSA) and million bits for Fully Homomorphic Encryption (FHE)). The computation of cryptographic algorithm required a lot of large integer arithmetic computation, especially large integer exponentiation and modular operation. Hence, large integer multiplication as the core operation of large integer modular exponentiation is the key factor in determining the performance of the cryptosystem in term of computation time.

The minimum bit size of encryption/decryption keys to achieve certain security level have increased over years. CPU implementations are becoming less efficient in handling the computation of crypto algorithms. Hence, other contemporary processor architectures such as GPU and FPGA have become popular alternative to speed up the computation in recent years.

This dissertation first discusses several existing large integer multiplication algorithms and reviews different methods used to implement the

discussed algorithms done by other researchers in both Graphic Processing Unit (GPU) and Field Programmable Gate Array (FPGA).

Compared to GPU, FPGA offered more low level design and development to the implementation. While GPU allowed users to configure each of the cores (processing units) to perform independent tasks, FPGA provides users a platform to design the processing units themselves to perform dedicated arithmetic and logic operation.

In our GPU implementation, we present two different large integer algorithms implementation on two generation of NVIDIA GPU architectures, Kepler and Pascal. The former focuses on utilizing the shuffle instructions to reduce memory latency and bulk multiplication that is able to perform up to 900 multiplications with operands size of 1024, 2048, 4096 and 8192-bit. This implementation also proved that our proposed method of utilizing the shuffle instructions feature to share data between registers memory is able to achieve up to 11.45% and 36.54% speedup when compared with conventional methods of storing data in GPU global memory and shared memory respectively; The later implementation on Pascal architecture aims to achieve single high speed multiplication of larger size by utilizing the available GPU resources. We are able to eliminate the bottleneck, the CRT algorithm from our previous implementation by replacing it with another level of CTFNT and hence increased the multiplication operand size to 4096, 192K, 384K and 768K-bit.

In our FPGA implementation, we focused on designing the processing unit ourselves that is capable of computing 3072-bit multiplication. We started with a preliminary design of a multiplier run with typical NTT module that perform computation in parallel-in-serial-out manner before we moved into design with radix-*R* FNT modules that are able to compute in parallel-in-parallel-out manner for better performance. Both radix-2 and radix-4 FNT modules are implemented and a further design of radix-4 FNT module, named radix-4 CTFNT module that is optimized to fit the multiplication algorithm we used is proposed. The proposed radix-4 design sacrificed the scalability for NTT with other parameters setup but is able to compute 16-points NTT with 10% faster performance and costs about 27% less resources to be implemented when compared to a typical radix-4 design.

## ACKNOWLEDGEMENTS

I wish to express my profound gratitude and sincere thanks to my supervisor, Prof. Goi Bok Min and co-supervisor Dr. Lee Wai Kong for their guidance in this research work, not only the technical skills but also encouragement they have given to me when I'm lost and lack of confidence. I would also like to say thank you to my lecturer Mr. Mok Kai Ming, who has been teaching me about hardware design since I am an undergraduate student. Besides, I would also like to extend my appreciation to my friends and lab members, especially Mr. Kiat Wei Pau, Mr. Tan Beng Liong, Mr. See Jin Chuan, Mr. Wong Xian Fu, Mr. Goey Jia Zheng, Mr. Phoon Jun Hoe and Mr. Siew Kar Hoe in sharing both their knowledge and experiences in GPU implementation and hardware developments. Last but not least, I would like to thank my parents for their understanding and supports for the decisions I made. Thank you for always be there for me, I would not have make it this far without you all.

This research work is partially supported by Ministry of Science, Technology and Innovation (MOSTI), Malaysia under grant 01-02-11-SF0202.

# APPROVAL SHEET

This dissertation entitled "**IMPLEMENTATION OF HIGH SPEED LARGE INTEGER MULTIPLICATION ALGORITHM ON CONTEMPORARY ARCHITECTURE**" was prepared by CHANG BOON CHIAO and submitted as partial fulfilment of the requirements for the degree of Master of Engineering Science at Universiti Tunku Abdul Rahman.

Approved by:

_____

(Ir. Prof. Dr. Goi Bok Min) Date:……

Supervisor

Department of Mechatronics and BioMedical Engineering

Lee Kong Chian Faculty of Engineering & Science

Universiti Tunku Abdul Rahman

_____

(Dr. Lee Wai Kong) Date: …….

Co-supervisor

Department of Computer and Communication Technology

Faculty of Information and Communication Technology

Universiti Tunku Abdul Rahman

## SUBMISSION SHEET

**LEE KONG CHIAN FACULTY OF ENGINEERING & SCIENCE**

**UNIVERSITI TUNKU ABDUL RAHMAN**

Date: _____

**SUBMISSION OF DISSERTATION**

It is hereby certified that **CHANG BOON CHIAO** (ID No: **16UEM01742**) has completed this dissertation entitled **"IMPLEMENTATION OF HIGH SPEED LARGE INTEGER MULTIPLICATION ALGORITHM ON CONTEMPORARY ARCHITECTURE"** under the supervision of **Ir. Prof. Dr. Goi Bok Min** (Supervisor) from the Department of Mechatronics and BioMedical Engineering, Lee Kong Chian Faculty of Engineering & Science, and **Dr. Lee Wai Kong** (Co-Supervisor) from the Department of Computer and Communication Technology, Faculty of Information and Communication Technology.

I understand that University will upload softcopy of my dissertation in pdf format into UTAR Institutional Repository, which may be made accessible to UTAR community and public.

Yours truly,

_____

(**CHANG BOON CHIAO**)

# DECLARATION

## DECLARATION

I Chang Boon Chiao hereby declare that the dissertation is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at UTAR or other institutions.

Name : CHANG BOON CHIAO

Date :

# TABLE OF CONTENTS

## LIST OF TABLES

**Table** **Page**

## LIST OF FIGURES

**Figures** **Page**

## LIST OF ABBREVIATIONS

| | |
|---|---|
| ASIC | Application-Specific Integrated Circuit |
| CLB | Configurable Logic Blocks |
| CPU | Central Processing Unit |
| CRT | Chinese Remainder Theorem |
| CTFFT | Cooley-Tukey FFT |
| DFT | Discrete Fourier Transform |
| DIF | Decimation in Time |
| DIT | Decimation in Frequency |
| DSP | Digital Signal Processor |
| GCD | Greatest Common Divisor |
| GMP | GNU Multiple Precision Arithmetic Library |
| GPU | Graphic Processing Unit |
| GTFFT | Good-Thomas Fast Fourier Transform |
| ECC | Elliptic Curve Cryptography |
| EDA | Electronic Design Automation |
| FFT | Fast Fourier Transform |
| FHE | Fully Homomorphic Encryption |
| FNT | Fast Number Theoretic Transform |
| FPGA | Field Programmable Gate Array |
| HDL | Hardware Description Language |
| IFFT | Inverse Fast Fourier Transform |
| INTT | Inverse Number Theoretic Transform |
| IoTs | Internet of Things |
| LUT | Look-Up Table |
| NTL | Number Theory Library |
| NTT | Number Theoretic Transform |
| NWT | Number Theoretic Weighted Transform |
| PKC | Public Key Cryptography |
| RAM | Random Access Memory |
| RSA | Rivest-Shamir-Adleman |
| SM | Streaming Multiprocessor |
| SSMA | Schönhage-Strassen Multiplication Algorithm |
| VHDL | VHSIC Hardware Description Language |
| VHSIC | Very High Speed Integrated Circuit |

# CHAPTER 1

# INTRODUCTION

## 1.1 Motivation

Information communication across the Internet requires security protocol to protect important and confidential data such as banking transactions and personal details from attackers/hackers. For decades, cryptosystems have been used to achieve this goal by performing data encryption, decryption and authentication during data transmission. Most of the cryptosystems are implemented with complex algorithms which require a lot of computing resources (i.e. memory size and processing power), resulting in slow performance.

Nowadays, Internet users are able to go online almost anytime from anywhere, thanks to the rapid growth of information and communication technologies. However, it is problematic if a cryptosystem is too slow to handle excessive requests from many users simultaneously. Leakage of secure information due to hackers' attack are also serious problems faced by many organizations employing modern cryptosystem to protect sensitive data. Thus, faster, safer and more reliable cryptosystems are on high demand.

The research for advanced cryptosystems hardware supports are heading towards new direction, from Central Processing Unit (CPU) to hardware accelerator such as Graphic Processing Unit (GPU) and Field Programmable

Gate Array (FPGA). This is because the architecture of GPU has more cores to perform multiple instructions simultaneously, which allowed faster computation for cryptographic algorithms. On the other hand, FPGA provides researchers to venture into techniques involving the design of the low level hardware modules.

Public Key Cryptography (PKC) which is also known as asymmetric-key cryptography where encryption and decryption are done using a pair of two different keys, a public key and a private key. PKC works in a way that the information is encrypted by sender with the recipient's public key (a key that can be accessed by anyone), but the encrypted information can only be decrypted by the recipient's paired private key. In this case, the transmission of information can be secured unless the recipient's paired private key is lost.

In Public Key cryptosystem (e.g. RSA), modular exponentiation is the bottleneck that costs most of the computational time, wherein multiplication is the core part of the modular exponentiation. The goals of this project are to research and develop techniques for accelerating the large integer multiplication, with implementation in GPU (software) and FPGA (hardware).

GPU is a popular platform to perform parallel computation, due to its massively parallel architecture (consists of multiple streaming multiprocessors (SMs)), which is capable in computing multiple instructions simultaneously. GPU also consists of deep memory hierarchy, with trade-off between memory size and

memory latency. Global memory has the largest capacity, followed by local memory, texture memory, constant memory, shared memory and registers. The global memory can be accessed by all threads within the same grid; the shared memory is limited to the threads within the same block while registers can only be used by a thread itself. However, started from NVIDIA GPU with Kepler or newer architecture, NVIDIA introduced a new feature known as "shuffle instruction" allowed registers to be visible by other threads within the same warp.

For FPGA implementation, a large integer multiplier hardware can be designed using Hardware Description Language (HDL) such as Verilog or VHDL. A designed hardware can be synthesized and implemented on development tool such as Xilinx Vivado for verification and validation. Different from GPU, hardware resources available on a FPGA board are used for the implementation. For example, Block Random Access Memory (BRAM) will be used to synthesize the memory elements; Look-up Table (LUT) are used to synthesize the internal hardware circuit; and Digital Signal Processors (DSP) can be used to speed up certain logic and arithmetic computation.

This project is divided into two phases. Phase one explores the possibilities of the efficient implementation of large integer multiplication in GPU, which involves investigation on the GPU architecture and implementation using parallel programming. Phase two focuses on hardware design and FPGA implementation of selected algorithm in phase one using Verilog HDL. The challenges include fitting the design into available resources on board, control

the data flow between sub-modules to meet desired functional behaviors and timing requirements.

## 1.2 Problem Statement

1. Modular Exponentiation is the most resource demanding operation in many Public Key Cryptography (e.g. RSA). The existing implementations of modular multiplication using Number Theoretic Transform (NTT) are not fully optimized for new GPU architectures.

2. Existing works of Modular Exponentiation in FPGA implementation only utilized the radix-2 architecture to implement NTT. The performance of other radices is not well studied, which may provide more opportunities for optimization.

## 1.3 Objectives

1. To research and optimize the implementation of large integer multiplication in state of the art GPU architectures.

2. To research and design a crypto-related processor capable in performing large integer multiplication in FPGA.

3. To propose new technique to optimize the performance of the designed crypto-related processor.

### 1.4 Contributions

In this research work, we presented two large integer multiplication implementations in GPU and one in FPGA:

1. The first implementation in GPU provides multiplication from 1024-bit to 8192-bit and up to 900 number of multiplications can be computed simultaneously, which can be used in ECC and RSA cryptographic algorithms. This implementation has proven that the method of utilizing the shuffle instructions, a new feature release from GPU with Kepler architecture onwards is able to reduce memory latency.

2. The second GPU implementation in this work is capable of computing large integer multiplication of 192K-bit, 384K-bit and 768K-bit, which is suitable to be used in toy version of Fully Homomorphic Encryption (FHE). This implementation shows a performance improvement in term of the operands size of the multiplication compared to the first implementation, with trading-off the ability of bulk multiplication.

3. A 3072-bit multiplier is implemented in FPGA. We compared the performance between radix-2 and radix-4 FNT module both theoretically and practically. We then proposed the idea of integrating radix-4 FNT modules that are optimized solely for the Cooley-Tukey FNT scheme we used in our 3K-bit multiplication algorithm in replace of standard radix-4 modules, which is able to reduce about 27% of the hardware resources.

### 1.5 Dissertation Organization

This chapter describes the motivation, problem statements, objectives and contributions of the project. The rest of the dissertation is organized as follows. Chapter 2 discusses about the background of our research including the algorithms and hardware platform; Chapter 3 presents the literature review on the similar research works done by the others; Chapter 4 describes the details of our implementation followed by the experimental setup and results in Chapter 5, and finally we conclude this project with conclusion and future works in Chapter 6.

# CHAPTER 2

# BACKGROUND

## 2.1 Graphic Processing Unit (GPU)

Graphic Processing Unit (GPU) is designed with highly parallel architecture which consists of multiple streaming multiprocessors (SMs) used to render the colors of pixels to create images and videos. In recent years, GPU has been employed to accelerate non-graphic computation including scientific computing, digital signal processing and intensive mathematical calculations.

### 2.1.1 GPU Programming Model

CUDA is a parallel computing platform and programming model with a small set of extensions to the C language developed by NVIDIA (Cheng et al., 2014). CUDA programming allows heterogeneous computing between CPU and GPU, where the CPU and its memory are defined as “Host” while the GPU and its memory is called “Device”. A typical execution on CUDA programming involved three steps:

1) Copy memory from host to device;
2) Invoke kernel(s) to execute program on device;
3) Copy memory back from device to host.

A thread is the basic element that processes data in a GPU kernel. CUDA organizes threads in two-level thread hierarchy, the grid and the block. Multiple

threads are grouped together to form a thread blocks; and a grid is made up of a number of thread blocks. To be specific, a grid is organized as a 2D array of blocks, and a block is organized as a 3D array of threads.

When a kernel is launched from host, a number of threads are spawned and each of them will executes the statements defined in the kernel function. During the execution, every 32 threads within a block are grouped into warps and each of these warps are then assigned to a streaming multiprocessor (the processing unit of GPU that consists of multiple cores) for execution. All of the threads within the same warp are implicitly synchronized.

### 2.1.2 GPU Memory Management

GPU architecture composes of deep memory hierarchy. Figure 2.1 illustrates different level of GPU memory, sorted from top to bottom is the memory level from largest to smallest memory size; while sorted from bottom to top is the memory level from shortest to longest memory latency.

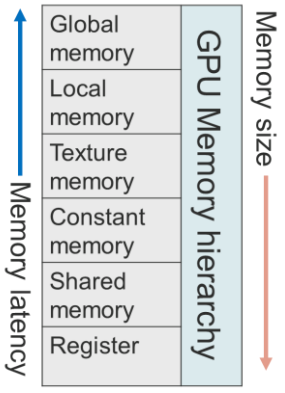


**Figure 2.1 GPU Memory**

Each of the GPU memory levels have their own constraints in threads accessibility. The global memory is GPU off-chip DRAM memory. It is accessible by all of the threads within the kernel; the shared memory is GPU on-chip memory, which the access is limited to the threads within the same blocks; while the register is built individually on each of the core of the GPU streaming multiprocessors, which can only be accessed by the thread itself. The texture memory and constant memory are cached and read-only memory in GPU where the constant memory can be used to reduce the required memory bandwidth compared to global memory, and the texture memory are used when all reads in a warp are physically adjacent. The local memory is similar to register, which is only limited to access by the thread itself but perform slower, typically same speed with global memory.

### 2.1.3 Shuffle Instruction

Starting with NVIDIA GPU with Kepler or newer generation of architecture and CUDA compute capability of 3.0 or higher, NVIDIA introduced a new feature named “Shuffle Instruction”. This new feature is a mechanism that allows the threads within the same warp to read the data stored in each other’s registers without the need of going through higher memory level, such as shared memory and global memory.

## 2.2 Field Programmable Gate Array (FPGA)

Field Programmable Gate Array (FPGA) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects (Xilinx, 2018). It is an integrated circuit that can be configured to meet desired application or functionality requirements after manufacturing. FPGA is commonly used as a development board for designers or engineers in digital system design and development phase.

### 2.2.1 FPGA Hardware Resources

The hardware resources available on a FPGA are one of the most important factor that have to take into consideration when designing a digital system. Theoretically, the more the resources are used the better the performance of the design. However, using a lot of resources require a large area of hardware. Resources overutilization will also cause the design to not be able to fit into the targeted FPGA board. The hardware resources available on a FPGA board includes: Look-Up Tables (LUTs), LUTRAM (LUT Random Access Memory), BRAM (Block RAM), Flip-Flops, Digital Signal Processors (DSPs) and input-output ports. The logics of a circuit are map into the LUTs when synthesized; the BRAM is the synchronous memory elements on board; the DSPs are used for fast arithmetic logic operation such as multiplication, normally used to speed up certain computation, especially the critical paths or used to reduce the number of LUTs used.

### 2.2.2 Hardware Description Language (HDL)

Hardware Description Language (HDL) is a specialized computer language used to describe the structure and behaviour of digital logic circuits. There are two types of HDL, the synthesis based HDL and the simulation based HDL. The synthesis based HDL is used to model digital logic circuit known as hardware module. A hardware module can also be modelled by connecting multiple hardware modules of lower level, defined as sub-modules; the simulation based HDL is a set of HDL statements that are not synthesizable, but for the aid of design verification. This set of HDL statements provide a set of virtual inputs to the designed hardware module, and the expected outputs can be viewed in the form of simulation waveforms or memory elements in simulation tools when simulated.

Two of the most commonly used HDLs are Verilog HDL and VHDL (VHSIC HDL, where VHSIC stands for Very High Speed Integrated Circuit). Verilog HDL is used in this research project.

### 2.2.3 FPGA Configuration

Configuring FPGA involves following steps: Started with hardware module modelling using HDL with functional behaviour verification of the modelled module; followed by the hardware synthesis before going into hardware implementation. Both hardware synthesis and hardware implementation required functional and timing behaviour verification to ensure the correctness of the design. The last step will be bitstream generation of the implemented design and the generated bitstream is downloaded into the FPGA board.

Hardware synthesis is the step that the Electronic Design Automation (EDA), is used to convert the designed hardware into LUTs, memory elements and other hardware resources available on the FPGA. Estimated values of required hardware resources will also be shown. EDA is the software program that assists in performing or automating design work. In this project, Xilinx Vivado HLx Edition 2016.3 is used. The hardware implementation step place and route the synthesized design to the targeted FPGA board virtually according to the design constraints file set in the EDA. The actual hardware resources utilization and power consumption are shown after the implementation. The timing behaviour simulation done after the hardware implementation will show the amount of time the signals or data required to go from one port to another. Once the implementation is done, the implemented design will be used to generate bitstream and load into the targeted FPGA board.

## 2.3 Modular Arithmetic and Large Integer Representation

### 2.3.1 Modular Arithmetic

Modular arithmetic refers to arithmetic operations (including addition, subtraction, multiplication and division) performed over a finite field and integer domain. The division is also known as multiplication of the dividend with the modular multiplicative inverse of the divisor. A finite field (also known as Galois field) is a field consists of a set of number ranged from 0 to $M - 1$, where $M$ is the modulus, the number used to generate the finite field.

In normal arithmetic, if the numbers a and b satisfied the equation $a / b = 1$, $b$ is the multiplicative inverse of $a$ ($a / b = a \times b^{-1}$). For example, 4 is the modular multiplicative inverse of 5 when $M = 19$ (5 x 4 mod 19 = 20 mod 19 = 1). In modular arithmetic, the modular multiplicative inverse exists if and only if the number, $a$ is coprime with the modulo, $M$. Two numbers are said to be coprime with each other if and only if 1 is the only positive common factor that divides both of them.

### 2.3.2 Large Integer Representation

In native computer systems, a single data supported is 64-bit. To represent a large integer, multiple 64-bit data are cascaded to form a large integer as shown in Figure 2.2. Each of these single data is known as a limb or precision.

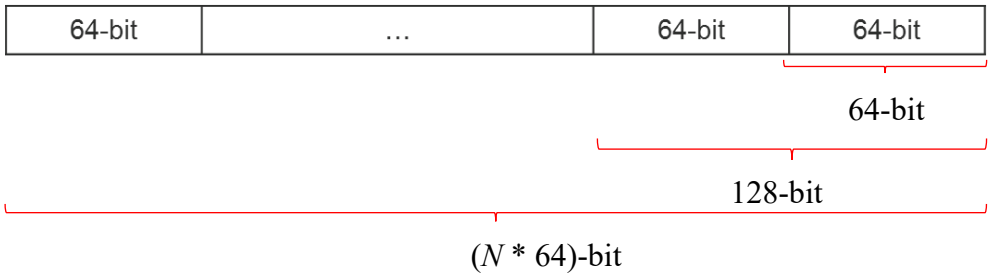


**Figure 2.2 Large integer representation**

Similar to polynomial representation of a number, where the radix, *R* is equal to $2^{64}$.

$$\mathrm{P}(R) = \sum_{i=0}^{N-1} a_i R^i \qquad (3.1)$$

**Table 2.1 Multiplication algorithms complexities**

| Multiplication algorithm | Complexity |
|---|---|
| Standard school book methods | $O(n^2)$ |
| Karatsuba's multiplication algorithm | $O(n^{log_2 3}) \approx O(n^{1.585})$ |
| Schönhage-Strassen's Multiplication Algorithm (SSMA) | $O(n \log n \log \log n)$ |

** $n$ is the number of data, or number of limbs.*

Table 2.1 shows three multiplication algorithms and their complexities.

SSMA utilizes FFT (Fast Fourier Transform) to achieve efficient multiplication.

## 2.4 Chinese Remainder Theorem (CRT)

Chinese Remainder Theorem stated that there is a unique solution for a number if this number produce several residue numbers when divided by *c* numbers of positive integers ($P_0, P_1, P_2, \ldots, P_{c-1}$), known as CRT moduli, with the following requirements:

1) These CRT must be co-prime to each other:

$$GCD(Pi, Pj) == 1, for\ i \neq j;\ 0 \leq i < c;\ 0 \leq j < c; \tag{3.2}$$

2) The solution number, $X * Y$ is smaller than the product of all of these CRT moduli:

$$X * Y < \prod_0^{c-1} P_i \tag{3.3}$$

A number, *X* can be converted to a series of coefficients, $a_0, a_1, a_2, \ldots, a_{c-1}$, which is the set of residue numbers after dividing *X* with the set of CRT moduli individually. This operation is known as Forward CRT while the process to reconstruct the original number from all of these residue numbers is called Inverse CRT.

Forward CRT:

$$a_i = X\ mod\ P_i, 0 \leq i < c \tag{3.4}$$

Inverse CRT:

$$X = \sum_{i=0}^{c-1} a_i\, m_i q_i\ mod\ \boldsymbol{P} \tag{3.5}$$

where $\boldsymbol{P} = \prod_0^{c-1} P_i$, $\quad m_i = \frac{\boldsymbol{P}}{P_i}$, $\quad q_i = {m_i}^{-1}\ mod\ P_i$

### 2.5 Schȍnhage-Strassen Multiplication Algorithm (SSMA)

Schȍnhage-Strassen Multiplication Algorithm is a well-known multiplication algorithm due to its low computational complexity of *O(n log n (log log (n)))*, where *n* is the number of input data. Standard school book multiplication algorithm is done in time domain, with a computational complexity of $O(n^2)$. SSMA allows the multiplication to be computed in frequency domain for better efficiency. The multiplication perform in frequency domain is known as convolution, which is point-wise multiplication that is able to achieve low computational complexity of *O(n).*

The performance of SSMA mainly rely on the transformation of the operands between time and frequency domain using NTT, which has a computation complexity of $O(n^2)$.

Pseudocode of SSMA:

Input : x and y, multi-precision large integers, the multiplier and multiplicand.

Output : z, multi-precision large integers, the product.

1) X ← NTT(x); //Forward NTT
2) Y ← NTT(y); //Forward NTT
3) Z ← CONVO(X, Y); //convolution
4) z ← INTT(Z); //Inverse NTT
5) evaluation(); //resolve the carries of z

A standard SSMA involved three NTTs (two forward transforms for both of the multiplication operands and one inverse transform for the product).

## 2.6 Fourier Transform

In electrical engineering and digital signals processing, Fourier Transform is known as a process to transform a signal from its time domain to frequency domain or vice-versa. The transformation from time domain to frequency domain is known as Forward Transform and transformation from frequency domain to time domain is called Inverse Transform. Fourier transform is computed using integral function, which performs on a continuous signal over a period of time.

### 2.6.1 Discrete Fourier Transform

Discrete Fourier Transform (DFT) is a discrete version of the Fourier transform, which collect *N* number of samples from the said continuous signal at a specific sampling rate. Same with Fourier Transform, DFT is computed in complex domain (operation involves imaginary and floating points number).

Forward Transform (DFT):

- $X_n = \sum_{k=0}^{N-1} x_k e^{-j\frac{2\pi}{N}nk}, \ 0 \leq n < N$

Inverse Transform (DFT):

- $x_k = \frac{1}{N}\sum_{n=0}^{N-1} X_n \, e^{j\frac{2\pi}{N}nk}, \ 0 \leq k < N$

$x$: series of DFT input elements
$X$: series of DFT output elements
$N$: number of DFT samples
$j$ : imaginary number
$e$ : Euler's constant

### 2.6.2 Fast Fourier Transform (FFT)

Fast Fourier Transform (FFT) is defined as a fast way to compute Discrete Fourier Transform (DFT). Typically improve the computational complexity from $O(N^2)$ to $O(N \log N)$, where $N$ is the number of FFT samples.

Multiple algorithms can be used to implement FFT. For example, the "Decimation-In-Time" (DIT) or "Decimation-In-Frequency" (DIF) FFT, Radix-*R* FFT, Good-Thomas FFT (GTFFT), and Cooley-Tukey FFT (CTFFT). There are also mixed-radix FFT and split-radix FFT, which are considered special cases of Radix-*R* FFT.

#### 2.6.2.1 Decimation-In-Time FFT and Decimation-In-Frequency FFT

A FFT algorithm can be computed by using "Decimation-In-Time" (DIT) or "Decimation-In-Frequency" (DIF) approaches. Both the DIT and DIF FFT employed divide-and-conquer technique to compute an *N*-point DFT in multiple DFTs of smaller size. The main difference between them is that DIT FFT divides *N* number of DFT points in old and even manner at each level whereas DIF FFT divides them into first-half and second-half of points at each level.

DIT Forward Transform (DFT):

$X_n = \sum_{k=0}^{N-1} x_k e^{-j\frac{2\pi}{N}nk}$, $0 \le n < N$; root of unity, $W = e^{-j\frac{2\pi}{N}}$

$= \sum_{k\in even}^{N-1} x_k W^{nk} + \sum_{k\in odd}^{N-1} x_k W^{nk}$

$= \sum_{k=0}^{\frac{N}{2}-1} x_{(2k)} W^{n(2k)} + \sum_{k=0}^{\frac{N}{2}-1} x_{(2k+1)} W^{n(2k+1)}$

$= \sum_{k=0}^{\frac{N}{2}-1} x_{(2k)} W^{n(2k)} + W^n \sum_{k=0}^{\frac{N}{2}-1} x_{(2k+1)} W^{n(2k)}$

DIF Forward Transform (DFT):

$X_n = \sum_{k=0}^{N-1} x_k e^{-j\frac{2\pi}{N}nk}$, $0 \le n < N$; root of unity, $W = e^{-j\frac{2\pi}{N}}$

$= \sum_{k=0}^{\frac{N}{2}-1} x_k W^{nk} + \sum_{k=\frac{N}{2}}^{N-1} x_k W^{nk}$

$= \sum_{k=0}^{\frac{N}{2}-1} x_k W^{nk} + \sum_{k=0}^{\frac{N}{2}-1} x_{(k+\frac{N}{2})} W^{n(k+\frac{N}{2})}$

$= \sum_{k=0}^{\frac{N}{2}-1} x_k W^{nk} + W^{n\frac{N}{2}} \sum_{k=0}^{\frac{N}{2}-1} x_{(k+\frac{N}{2})} W^{nk}$

### 2.6.2.2 Radix-*R* FFT

A radix-*R* FFT decomposes an *N*-point DFT into *N*/*R* number of *R*-point DFTs through $log_R N$ levels of decomposition. Given *l* is equal to the index of the level and the index of the top level DFT (before any decomposition) is equal to 0 with increment of 1 at each level of decomposition, the number of DFTs at each level is equal to $R^l$; the number of points in one DFT is equal to $N/R^l$. Note that for an *N*-point DFT to be implemented with radix-*R* FFT, *N* must be a power of *R*. A method known as "mixed-radix FFT" can be used to overcome this issue when *N* is not a power of *R* by performing different radix of FFT decomposition at different level.

Figure 2.3 shows the 2-point FFT structures (a.k.a. the butterfly structures), the smallest module of a radix-2 FFT for both DIT and DIF FFT where $ip_0$ and $ip_1$ are the input to the FFT; $op_0$ and $op_1$ are the output to the FFT. The **W* indicates the twiddle factors multiplication of the FFT. The different between radix-2 DIT FFT and radix-2 DIF FFT is when the twiddle factor multiplication to the second input, $ip_1$ is performed. The former performs the multiplication before the crossing add-and-subtract operation while the later do it after the crossing add-and-subtract operation.

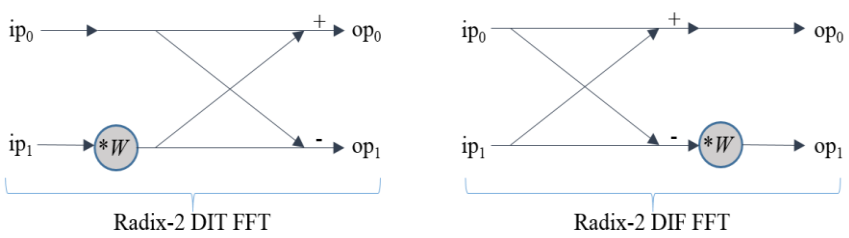


**Figure 2.3 Radix-2 DIT and DIF 2-point FFT modules**

Both the DIT and DIF FFT as discussed in previous section (section 2.6.2.1) can be implemented with radix-*R* FFT. Figure 2.4 shows how multiple radix-2 FFT modules can be connected together to compute 4-point FFT, where the diagram on the left is implemented with DIT FFT structure while the one on the right is built with DIF FFT. Figure 2.5 and Figure 2.6 illustrate the flow of 8-point FFT implemented with radix-2 DIT FFT and DIF FFT respectively. The dashes boxes indicate the radix-2 modules.

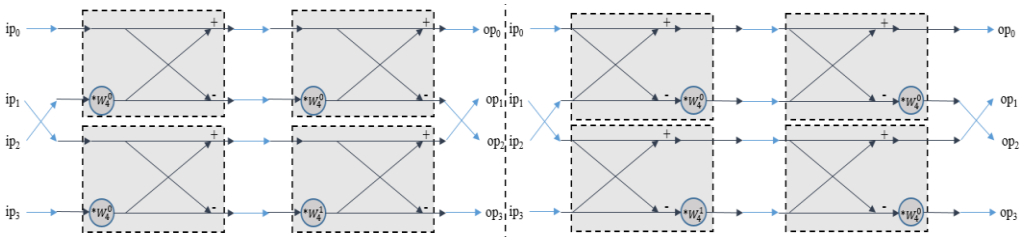


**Figure 2.4 4-point Radix-2 DIT and DIF FFT comparison**

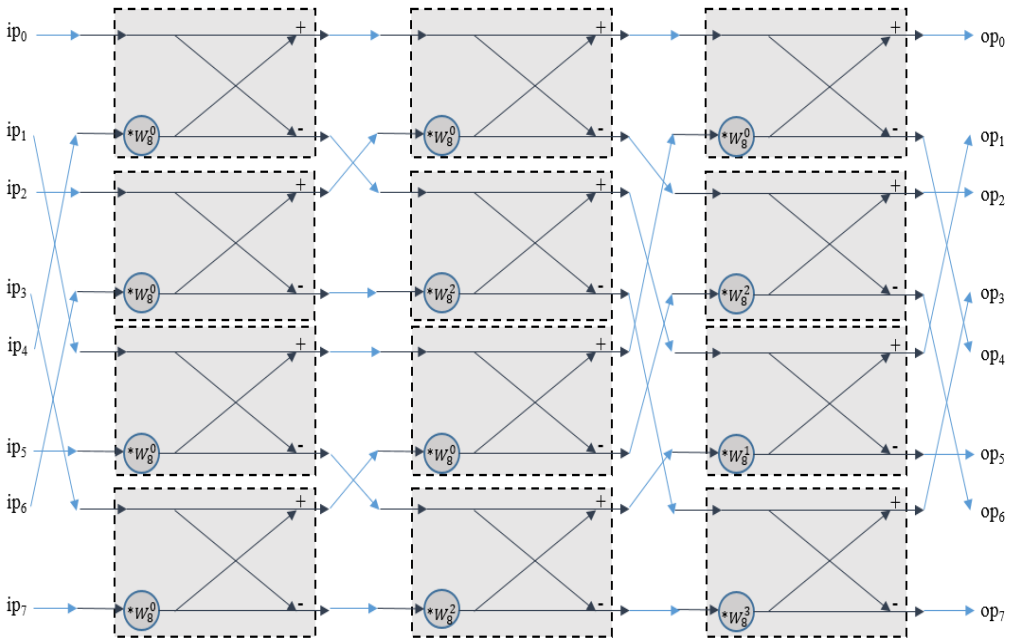


**Figure 2.5 8-point Radix-2 DIT FFT**

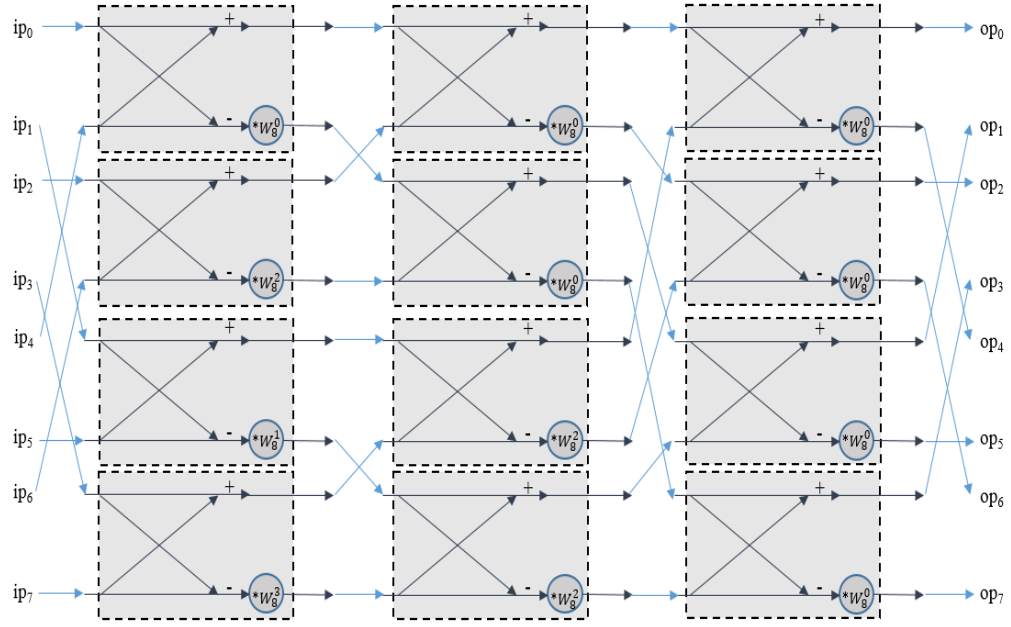


**Figure 2.6 8-point Radix-2 DIF FFT**

Equation below derived the radix-4 DIF FFT from radix-2 DIF FFT.

DIF Forward Transform (DFT):

$X_n = \sum_{k=0}^{N-1} x_k e^{-j\frac{2\pi}{N}nk}$, $0 \le n < N$; root of unity, let $W = e^{-j\frac{2\pi}{N}}$

$$= \sum_{k=0}^{\frac{N}{2}-1} x_k W^{nk} + W^{n\frac{N}{2}} \sum_{k=0}^{\frac{N}{2}-1} x_{(k+\frac{N}{2})} W^{nk}$$

$$= \sum_{k=0}^{\frac{N}{4}-1} x_k W^{nk} + \sum_{k=\frac{N}{4}}^{\frac{2N}{4}-1} x_k W^{nk} + \sum_{k=\frac{2N}{4}}^{\frac{3N}{4}-1} x_k W^{nk} + \sum_{k=\frac{3N}{4}}^{N-1} x_k W^{nk}$$

$$= \sum_{k=0}^{\frac{N}{4}-1} x_k W^{nk} + \sum_{k=0}^{\frac{N}{4}-1} x_{(k+\frac{N}{4})} W^{n(k+\frac{N}{4})} + \sum_{k=0}^{\frac{N}{4}-1} x_{(k+\frac{2N}{4})} W^{n(k+\frac{2N}{4})} +$$

$$\sum_{k=0}^{\frac{N}{4}-1} x_{\left(k+\frac{3N}{4}\right)} W^{n\left(k+\frac{3N}{4}\right)}$$

$$= \sum_{k=0}^{\frac{N}{4}-1} x_k W^{nk} + W^{n\frac{N}{4}} \sum_{k=0}^{\frac{N}{4}-1} x_{(k+\frac{N}{4})} W^{nk} + W^{n\frac{N}{2}} \sum_{k=0}^{\frac{N}{4}-1} x_{(k+\frac{N}{2})} W^{nk} +$$

$$W^{3n\frac{N}{4}} \sum_{k=0}^{\frac{N}{4}-1} x_{(k+\frac{3N}{4})} W^{nk}$$

Figure 2.7 shows the structures of radix-4 FFT. The structure on the left is a detailed version of a single radix-4 FFT module while the one on the right is a simplified version. The internal operations are omitted for simplicity.

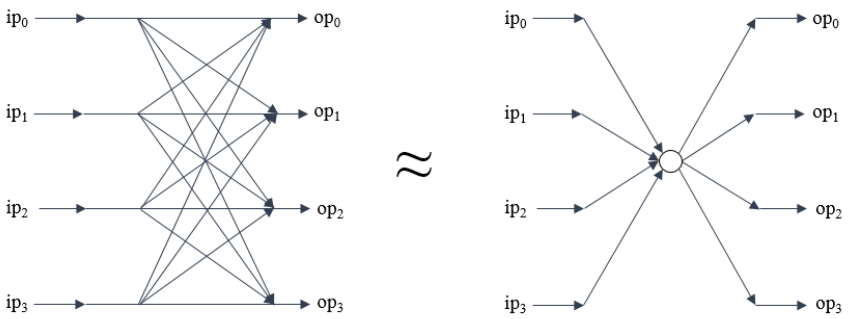


**Figure 2.7 radix-4 FFT module**

### 2.6.2.3 Good-Thomas FFT (GTFFT)

Good-Thomas FFT (GTFFT) (a.k.a. Prime Factor FFT) is a FFT algorithm that lets $N = N_1 * N_2$ and breaks an $N$-point DFT into ($N_1 * N_2$)-point DFT, where $N_1$ and $N_2$ have to be relatively prime. The ($N_1 * N_2$)-point DFT can be computed in two steps, started with $N_2$ number of $N_1$-point DFT followed by $N_1$ number of $N_2$-point DFT. Hence, the GTFFT is sometimes defined as a FFT algorithm that compute a DFT in two-dimensional FFT.

The GTFFT equation can be derived by substituting the following equation into the DFT forward transform equation:

- $n = N_2 n_1 + N_1 n_2 \; mod \; N, \quad 0 \leq n_1 < N_1, 0 \leq n_2 < N_2$
- $k = {N_2}^{-1} N_2 k_1 + {N_1}^{-1} N_1 k_2, \; 0 \leq k_1 < N_1, 0 \leq k_2 < N_2$

Where:

${N_1}^{-1}$ denotes the modular multiplicative inverse of $N_1 mod \; N_2$ and

- ${N_2}^{-1}$ denotes the modular multiplicative inverse of $N_2 mod \; N_1$.

Forward Transform (GTFFT):

- $X_{{N_2}^{-1} N_2 k_1 + {N_1}^{-1} N_1 k_2} = \sum_{n_1=0}^{N_1-1} ( \sum_{n_2=0}^{N_2-1} x_{N_2 n_1 + N_1 n_2} e^{-j \frac{2\pi}{N_2} n_2 k_2} ) e^{-j \frac{2\pi}{N_1} n_1 k_1}$

Inverse Transform (GTFFT):

- $x_{{N_2}^{-1} N_2 k_1 + {N_1}^{-1} N_1 k_2} = \sum_{n_1=0}^{N_1-1} ( \sum_{n_2=0}^{N_2-1} X_{N_2 n_1 + N_1 n_2} e^{-j \frac{2\pi}{N_2} n_2 k_2} ) e^{-j \frac{2\pi}{N_1} n_1 k_1}$

#### 2.6.2.4 Cooley-Tukey FFT (CTFFT)

CTFFT is a FFT algorithm that allows a large size DFT to be computed with multiple DFTs of smaller size, improving parallelism. Similar to GTFFT, CTFFT uses a special indexing method to access the large size DFT in two dimensions' manner by dividing the DFT size (let $N = N_1 * N_2$), except $N_1$ does not necessary have to be coprime with $N_2$. It is also sometimes described as a FFT algorithm that compute a single row $N$-points DFT in two dimensional row-by-column, ($N_1 * N_2$)-points DFTs, with $N_1$ number of rows and $N_2$ number of columns, where $N = N_1 * N_2$. Generally, CTFFT consists of three steps:

1) Row FFT ($N_1$ number of $N_2$ points DFT);
2) Twiddle factors multiplication;
3) Column FFT ($N_2$ number of $N_1$ points DFT);

Forward Transform (CTFFT):

- $X_{j_1+N_1 j_2} = \sum_{i_2=0}^{N_2-1}[(\sum_{i_1=0}^{N_1-1} x_{N_2 i_1+i_2} e^{-j\frac{2\pi}{N_1} i_1 j_1}) e^{-j\frac{2\pi}{N} i_2 j_1}] e^{-j\frac{2\pi}{N_2} i_2 j_2}$

Inverse Transform (CTFFT):

- $x_{j_1+N_1 j_2} = N^{-1} \sum_{i_2=0}^{N_2-1}[(\sum_{i_1=0}^{N_1-1} X_{N_2 i_1+i_2} e^{-j\frac{2\pi}{N_1} i_1 j_1}) e^{-j\frac{2\pi}{N} i_2 j_1}] e^{-j\frac{2\pi}{N_2} i_2 j_2}$

The above equations can be derived by substituting the following equation into the equation in NTT:

- $i = N_2 i_1 + i_2, 0 \leq i_1 < N_1, 0 \leq i_2 < N_2$
- $j = N_1 j_2 + j_1, 0 \leq j_1 < N_1, 0 \leq j_2 < N_2$

### 2.6.3 Number Theoretic Transform (NTT)

Number Theoretic Transform (NTT) is a mathematics function that transform a set of data between its time and frequency domain. Same with Fourier transform and DFT, transformation of time-to-frequency domain is called Forward Transform and frequency-to-time domain transformation is known as Inverse Transform. This transformation is performed over a finite field.

Forward Transform (NTT):

- $X_j = \sum_{i=0}^{N-1} x_i g^{ij} \, mod \, M, \;\; 0 \leq j < N$

Inverse Transform (NTT):

- $x_i = N^{-1} \sum_{j=0}^{N-1} X_j g^{ij} \, mod \, M, \;\; 0 \leq i < N$

$x$: series of NTT input elements
$X$: series of NTT output elements
$N$: NTT size, number of elements to be transformed
$g$: primitive N$^{th}$ root of unity
$M$ : modulus

The $N^{th}$ root of unity, $r$, is a number that when multiply itself $N$ times will yield 1 in a finite field, $r^N \equiv (1 \bmod M)$, where $M$ is the modulus used to generate the finite field and $r$ must be larger than 1 and smaller than $M$ ($1 < r < M - 1$). For the $N^{th}$ root of unity to be primitive, $r$ must be the smallest positive integer found within the range.

Pseudocode of NTT:

Input : 1) $x$, series of NTT input elements;
2) $tf$, series of precomputed twiddle factors;

Output : $X$, series of NTT output elements;

tmp; //temporary value;

1) for j := 0 to $N$ - $1$ do
2) $X_j$ := 0
3) for i := 0 to $N$ - $1$ do
4) tmp ← $x_i$ * $tf_{ij}$ mod $M$
5) $X_j$ ← $X_j$ + tmp mod $M$
6) end
7) end

### 2.6.4 Fast Number Theoretic Transform (FNT)

Both DFT and NTT shared the same definition: the output data with index *i*, is equals to the summation of products from the input data with their respective twiddle factors. The main difference between both of the transformation is, the DFT is computed in complex domain that involved floating-point arithmetic while NTT is performed in integer domain of finite fields, computed with modular arithmetic. Thus, the FFT algorithm can also be applied on NTT to speed up the computation.

In subsequent discussions, we refer FFT as a fast way to compute DFT and denote Fast Number Theoretic Transform (FNT) as fast way to compute NTT for the rest of the dissertation. For instance, the CTFFT algorithm can be modified to become CTFNT by replacing the root of unity in DFT, $e^{-j\frac{2\pi}{N}}$, with the root of unity of NTT, $g_N$ and changing the computation domain from complex domain to integer finite field domain.

Forward Transform (CTFNT):

- $X_{j_1+N_1 j_2} = \sum_{i_2=0}^{N_2-1}[(\sum_{i_1=0}^{N_1-1} x_{N_2 i_1+i_2} {g_{N_1}}^{i_1 j_1}) {g_N}^{i_2 j_1}] {g_{N_2}}^{i_2 j_2} \ mod\ M$

Inverse Transform (CTFNT):

- $x_{j_1+N_1 j_2} = N^{-1} \sum_{i_2=0}^{N_2-1}[(\sum_{i_1=0}^{N_1-1} X_{N_2 i_1+i_2} {g_{N_1}}^{i_1 j_1}) {g_N}^{i_2 j_1}] {g_{N_2}}^{i_2 j_2} \ mod\ M$

Pseudocode of CTFNT:

Input : 1) $x$, series of NTT input elements;
2) $t$, series of temporary intermediate value;
3) $tfN0$, series of precomputed twiddle factors for $N$;
4) $tfN1$, series of precomputed twiddle factors for $N_1$;
5) $tfN2$, series of precomputed twiddle factors for $N_2$;

Output : $X$, series of NTT output elements;

tmp; //temporary value;

```
        //column NTTs
1)  for i2 := 0 to N2 - 1 do
2)      for j1 := 0 to N1 - 1 do
3)          t(i2 * N1 + j1) := 0
4)          for i1 := 0 to N1 - 1 do
5)              tmp ← x(i1 * N2 + j1) * tfN1(i2 * i1) mod M
6)              t(i2 * N1 + j1) ← t(i2 * N1 + j1) + tmp mod M
7)          end
8)      end
9)  end

        //twiddle factors multiplication
10) for i2 := 0 to N2 - 1 do
11)     for j1 := 0 to N1 - 1 do
12)         t(i2 * N1 + j1) ← t(i2 * N1 + j1) * tfN0(i2 * j1) mod M
13)     end
14) end

        //row NTTs
15) for i1 := 0 to N1 - 1 do
16)     for j2 := 0 to N2 - 1 do
17)         t(i1 * N2 + j2) := 0
18)         for i2 := 0 to N2 - 1 do
19)             tmp ← t(i1 * N2 + j1) * tfN2(i2 * i1) mod M
20)             X(i1 * N2 + j1) ← X(i1 * N2 + j1) + tmp mod M
21)         end
22)     end
23) end
```

Figure 2.8 shows the flowchart for the CTFNT in a simple flow which included three sub-processes, the column NTTs, twiddle factors multiplication and row NTTs. The flowchart for twiddle factors multiplication is illustrated in Figure 2.9 while the flowcharts of both columns NTTs and row NTTs are shown side-by-side in Figure 2.10 for better comparison. Note that the flowcharts of both column NTTs and row NTTs are similar with the difference of the indices used.

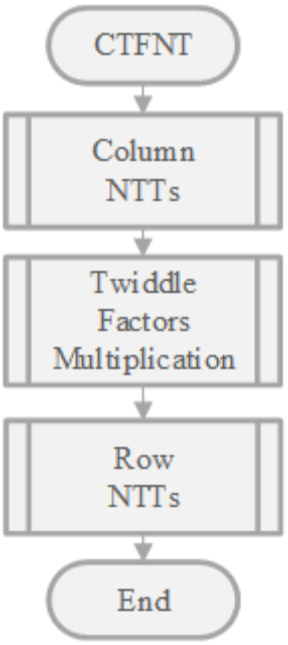


**Figure 2.8 Flowchart of CTFNT**

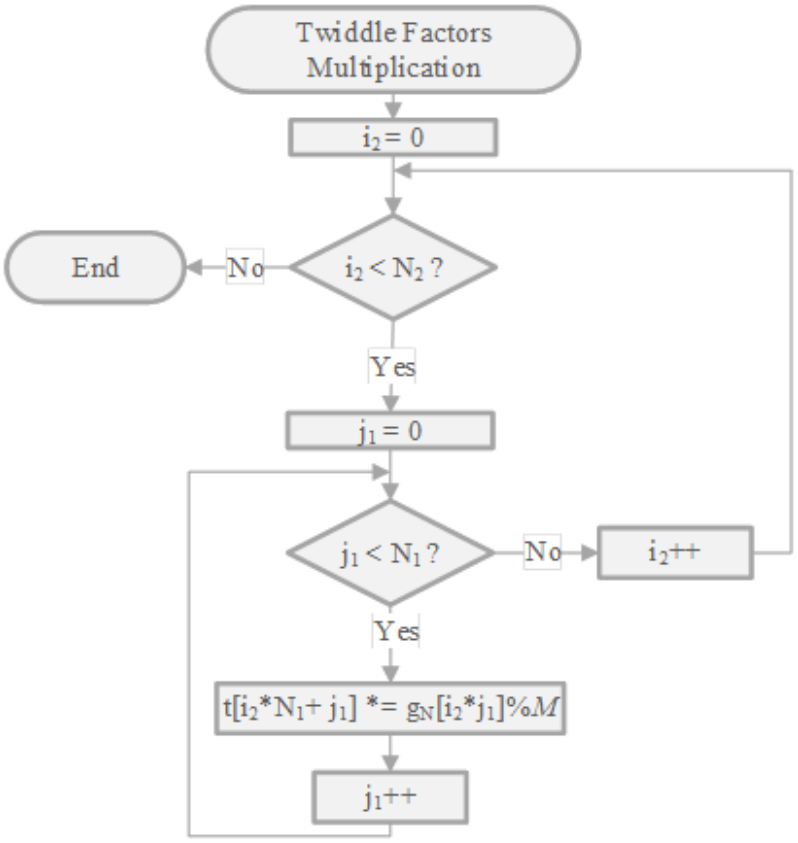


**Figure 2.9 Flowchart of Twiddle Factors Multiplication**

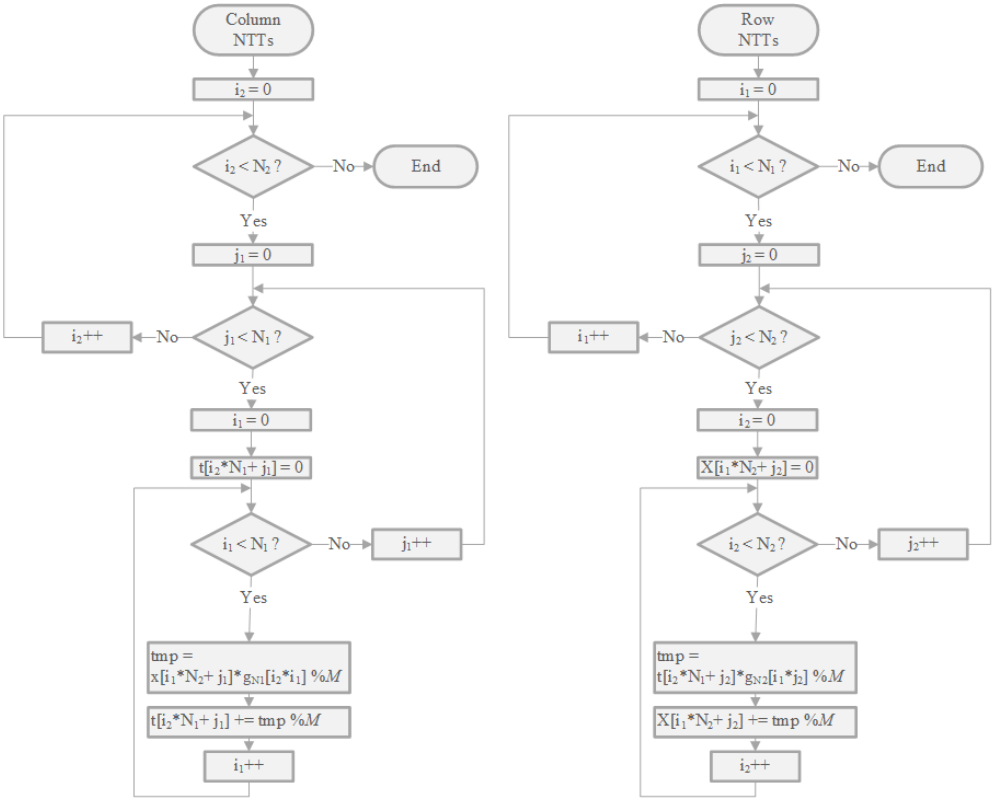


**Figure 2.10 Flowcharts of Column NTTs and Row NTTs**

## 2.7 Summary

The study between GTFFT and CTFFT shows that although GTFFT does not require twiddle factors multiplication in between row FFTs and column FFTs, its transformation size is limited to only relatively prime numbers, which will affect the effectiveness of balance workload distribution in implementation. In this case, CTFFT is more suitable as the transformation size is more dynamic.

Similar to radix-*R* FFT, CTFFT employs divide-and-conquer method to compute a full FFT. The main difference between them is radix-*R* FFT divides the *N*-FFT into *R* number of FFTs at each level until each of the FFTs have only *R* number of data; while CTFFT is more dynamic that divides a *N*-points FFT into ($N_1$ * $N_2$)-points FFT, where $N = N_1 * N_2$. CTFFT can also be used recursively to achieve higher level of parallelism.

Comparing radix-2 and radix-4 FFT, radix-2 FFT module is smaller and simple compare to radix-4 FFT module, and radix-4 FFT module also have longer latency compare to radix-2 FFT module. However, since a radix-*R* FFT module computes *R* number of data at each level, a radix-4 FFT module is able to compute more data than a radix-2 FFT module at each level. Hence, implementing radix-4 FFT module effectively can help to reduce the number of read and write access to data memory at the costs of extra hardware resources and slightly more complex control logic compare to radix-2 FFT module. A radix-4 FFT module can also be implemented by connecting four radix-2 FFT modules in two-by-two manner.

Theoretically, a radix-*R* FFT module with higher value of *R* is able to compute more data at each level, reducing the number of levels and hence reduce the number of memory access. However, increasing the value of *R* will also reduce the level of parallelism. The improvement of reducing the number of levels is also diminishing as the value of *R* grow higher.

In this work, NTT is more suitable to be used in SSMA compared to DFT as only integer domain is involved in SSMA. Using DFT will also introduce round-off errors when the transformation size increased. Different FFT algorithms can be used in different levels of the FFT decomposition. In our implementation, we focus on combining CTFNT with radix-4 FNT for SSMA.

Comparing GPU and FPGA, GPU is considered low costs for implementation but FPGA is able to perform faster computation. NVIDIA GPU provides CUDA, a programming platform that allowed the designers or programmers to configure kernel and manage different level of memory conveniently and systematically; the FPGA design platform is more scalable as the designers are able to design the system processing units themselves.

# CHAPTER 3

# LITERATURE REVIEW

## 3.1 Implementation in GPU

Graphic Processing Unit (GPU) is known for its highly parallel architecture with thousands of cores (processing units) to compute the color pixels of images and rendering video. However, the idea of employing GPU to accelerate various general purpose algorithms (typically run in CPU) such as scientific computing and cryptosystems has becoming popular in recent years. The research works done by (Emmart et al., 2011; Emeliyanenko et al., 2009; and T. Honda et al., 2015) in exploiting GPU to compute large integer multiplication had shown significant improvements over CPU implementation.

Both of the research works by Emmart and Weems (2011) and Emeliyanenko (2009) employed Schönhage-Strassen Multiplication Algorithm (SSMA) in their implementation. The works done by Emeliyanenko (2009) is able to outperform NTL and GMP (libraries for large integer multiplication) implementation in CPU with 512, 1024, and 2048-bit multiplication. Emmart and Weems (2011) integrated Cooley-Tukey FFT with Swartauber and Stockham indexing method into their SSMA implementation, achieving a maximum speedup of 19 times with multiplication operands ranged from 255K-bit to 16320K-bit when running the implementation on NVIDIA GTX480 GPU and Intel Core i7 870.

GPU with newer architecture comes with new features can be used to improve these works. For example, starting from NVIDIA Fermi GPU architecture, 64-bit multiplication is supported from high-level-language programmer's perspective, eliminated the needs of breaking operands into residue numbers of 24-bit to utilize native multiplication as in the work done by Emeliyanenko (2009). Besides, from NVIDIA Kepler onwards, new “shuffle instruction” is introduced to all new GPU, which allowed threads within the same warp to read registers from each other without the needs of going into higher memory level can help to reduce the needs of access to shared memory as done in the work of Emmart and Weems (2011).

The work done by T. Honda et al. (2015) utilized the “shuffle instructions” feature to perform bulk multi-precision integer multiplication (compute 100,000 multiplication simultaneously) ranged from 1024-bit to 32,768-bit with three different multiplication algorithms: Comba’s multiplication, Karatsuba’s multiplication and recursive Karatsuba’s multiplication is able to achieve a maximum speedup factor of 62.88 for 1024-bit multiplication and 18.71 times speedup for 32,768-bit multiplication. The comparison is done with NVIDIA GeForce GTX 980 with GNU multi precision library (GMP) running in Intel Xeon X7460 CPU.

W. Dai et al. (2015) presented a CUDA GPU library for Homomorphic encryption (cuHE) in their work. The implementation use GPU and interface with NTL library to compute Number Theoretic Transform (NTT) and Chinese

Remainder Theorem (CRT) for the applications of Prince block cipher and homomorphic sorting algorithms. Implemented on NVIDIA Geforce GTX 680, GTX770 and GTX690, their work has shown a 40 times speedup on a single GPU, 135 times on three GPUs simultaneously over the work done in CPU by Doröz et al. (2014). This work also able to show a speedup of 25 times when compared to Dai et al.'s work (2014) implemented on the same GPU device. The authors also utilized the multi-streams method to speed up the memory copy operations between CPU and GPU. There is no result of multiplication shown in the works, but there is possibility of using the NTT function provided in this library to implement SSMA.

### 3.2 Implementation in FPGA

Aside of GPU, FPGA is an alternative hardware platform that can be designed to perform dedicated algorithms. W. Dai et al. (2016) 's researches and W. Dai et al. (2017) have shown a comprehensive studies of Montgomery modular multiplication implementation on FPGA based on SSMA. Designed with Verilog-HDL and implemented with Xilinx ISE 13.3 design tool on Xilinx Virtex-6 (xc6vlx130t-1) FPGA board, the research work done by Chen et al. (2016) presented multiplication with operands size ranged from 1K-bit to 4K-bit. The work is done with different parameters set to show the trade-off between faster computation and smaller area. The authors also proposed a novel method known as "carry-save arithmetic" to improve parallelism for the resolve carries part of the SSMA, achieving 3100-bit and 4124-bit modular multiplications in 6.74 and 7.78us respectively. This method outperformed previous state-of-the-art work by Huang et al. (2011) for operand size of 3072-bit and above in term of computation latency and area-latency. FPGA design are often limited by the hardware resources such as the Lookup Table (LUTs) and memory available on board. Implementation with fully *N*-point NTT butterfly structures could lead to resources over-utilized; while using a single NTT butterfly structure repeatedly could help to minimize the resources required but on the other hand increased the latency. The work done by Chen et al. (2016) here utilized double radix-2 NTT modules in their implementation, but the use of a single radix-4 NTT module is yet to explore.

W. Dai et al. (2017) have further improved their SSMA over their previous work by Chen et al. (2016) by replacing Number Theoretic Transform (NTT), the most

crucial part of SSMA with Number Theoretic Weighted Transform (NWT). This method can efficiently reduce the convolution length in the SSMA by half, with an additional multiplication to the "weight", a set of extra parameters for each of the elements within the NTT. Next, the authors integrate the Montgomery Modular Multiplication into McLaughlin's framework, achieving better area-time efficiency compared to their previous work (Chen et al., 2016) (50.9% for 1,024-bit, 41.9% for 2,048-bit, 37.8% for 4,096-bit and 103.2% for 7,680-bit multiplication).

The work done by Chen et al. (2015) replaced acyclic convolution in SSMA with negative wrapped convolution. This method removes the needs of zero-padding half of the NTT elements required for typical SSMA, reduced the transformation size of NTT by half, which is similar to the method used by Dai et al. (2017) as described above. Same with the work by Chen et al. (2016), this work is implemented with two radix-2 NTT modules as their smallest NTT processing unit. However, each of the radix-2 NTT modules is used to compute forward transform for both of the input operands independently at the same time instead of using both of the modules to compute the first input operand followed by the second. The authors also implemented their design with pipeline architecture to increase module throughput and support higher frequency. Implemented in Spartan-6 (xc6slx100-3) FPGA using Verilog with Xilinx ISE14.7, the authors are able to achieve an average of 3.5 times speedup when compared to the work done by Pöppelmann and Güneysu (2012).

Targeted for the uses of Fully Homomorphic Encryption (FHE), Y. Doröz et al. (2014) published a work that is able to compute very large multiplication of two 1,179,648-bit operands in 7.74ms, synthesized in Synopsis Design Compiler with TSMC 90nm cell library. Similar to the works done by (Chen et al., 2016; Dai et al., 2017), SSMA is used in the implementation with the difference of using Cooley-Tukey FFT decomposition to reduce the computation time of NTT instead of fully radix-2. However, the implementation is done with 98,304-point NTT, a non-power-of-2 NTT transformation will affect the level of parallelism.

### 3.3 Summary

Crypto algorithms such as ECC (Elliptic Curve Cryptography), RSA (Rivest-Shamir-Adleman) and FHE (Fully Homomorphic Encryption) require intensive use of large integer arithmetic operations, from 512-bit (ECC), 2048-bit (RSA) up to millions-bit (FHE). Out of all arithmetic operations, modular exponentiation is the operation that costs the most computation time with multiplication as the core operation.

Multiplication algorithms including Comba's multiplication, Karatsuba's multiplication and especially SSMA have been widely studied and implemented with various methods across different platforms using GPU, FPGA and Application-Specific Integrated Circuit (ASIC) in the works of (Emmart and Weems, 2011; Emeliyanenko, 2009; Honda, Ito and Nakano, 2015; Dai and Sunar, 2015; Dai et al., 2014; Chen et al., 2016; Dai et al., 2017; Huang et al., 2011; Doröz et al., 2014; Chen et al., 2015; Pöppelmann and Güneysu, 2012; Huang and Wang, 2015) in order to achieve faster computation time, higher throughput, lesser hardware resources or better area-time efficiency.

The computational complexity for standard schoolbook's and Comba's multiplication is $O(n^2)$ and $O(n^{log_2 3})$ for the Karatsuba's multiplication algorithm, where $n$ is the number of precisions used to form the large integer. The Karatsuba's method reduces the number of internal multiplications between different precisions of the multiplicand and multiplier operands, at the costs of extra number of additions, which are cheaper operations in term of computation.

Implementing the Karatsuba's multiplication algorithm recursively can further reduce the number of internal multiplication. However, the work done by T. Honda et al. (2015) showed that applying Karatsuba's multiplication algorithm at each level will also increases the number of read and write memory access, which introduce extra overhead in GPU implementation. Hence, optimizing the algorithm to achieve lower computation complexity alone is not enough. Taking the hardware specifications into consideration is crucial as well.

SSMA appeared to be the most commonly used algorithms for large integer multiplication as it has the lowest computational complexity of *O(n log n log log n)* among the others. This algorithm is introduced by Schönhage and Strassen (1971), which utilized the FFT to perform point-wise multiplication in frequency domain of the number to achieve such a low computational complexity. Hence, it is also sometimes being called FFT-based multiplication or FFT-based polynomial multiplication.

However, performing FFT and inverse FFT in this algorithm incurred overhead. The work done by Baktır and Sunar (2006) stated that the integer has to be at least 1000-bit in order to compensate the overhead incurred. This value can be vary due to the algorithms used to implement the FFT and hardware specification.

SSMA is being implemented in the research works: (Emmart and Weems, 2011; Emeliyanenko, 2009; Dai and Sunar, 2015; Chen et al., 2016; Dai et al., 2017;

Chen et al., 2015; Huang and Wang, 2015), using different algorithms to compute FFT and altering the parameters set of the FFT to get the most optimum results. The works by Chen et al. (2016) and Dai et al. (2017) even integrated SSMA into Montgomery modular multiplication in replaced of traditional interleaved method and showed a significant improvement in their implementation over previous works.

# CHAPTER 4

# IMPLEMENTION DETAILS

## 4.1 Modular Arithmetic Functions

The data type UINT64 (unsigned long long int), 64-bit unsigned integer, and the largest bit size of single precision supported to run in GPU is used for the implementation of NTT in this work. To avoid overflow or underflow during the arithmetic operations, three modular functions are implemented with the modulus, $P$ = 0xFFFF_FFFF_0000_0001 as follows:

- UINT64 _addModP(UINT64 in1, UINT64 in2) ; //Modular addition
- UINT64 _subModP(UINT64 in1, UINT64 in2) ; //Modular subtraction
- UINT64 _mulModP(UINT64 in1, UINT64 in2) ; //Modular multiplication

### 4.1.1 Modular addition

The implementation of the modular addition can be done by first adding both the inputs and check if the result is smaller than the first input. If yes, overflow happened and adding additional value of 0xFFFF_FFFF to the result can help to recover the missing value in the finite field of $P$. Lastly, check if the result is greater or equal to $P$. If yes, return the result with $P$ subtracted from it else return the result untouched.

### 4.1.2 Modular subtraction

For the modular subtraction function, subtract the second input (subtrahend) from the first input (minuend). Check if the underflow condition happened by comparing the result with the minuend. If the result is greater than the minuend, underflow occurred. Adding the result with *P* to recover the actual result in the finite field. Lastly, check if the result is greater or equal to *P*. If yes, return the result with *P* subtracted from it else return the result untouched. This function is also used to make sure no negative value will be incurred throughout the entire program.

### 4.1.3 Modular multiplication

The implementation for the modular multiplication is more complex compared to addition and subtraction as multiplying two data of 64-bit each yield a result of 128-bit, which is way more than a single precision can handle. In general, any number can be represented in its polynomial form, which is the summation of a series of coefficients multiply with its radix to the power of its respective index, $A(\mathrm{r}) = \sum_{i=0}^{N-1} a_i r^i$, where *A* is the number, *r* is the radix and $a_i$ is the set of coefficients. Similarly, An 128-bit number can be represented in four coefficients of radix, $r = 2^{32}$, setting all the coefficients at 32-bit each, as shown below:

$$A_{128-bit}(2^{32}) = a_3 r^3 + a_2 r^2 + a_1 r^1 + a_0 r^0$$

$$A_{128-bit}(2^{32}) = a_3(2^{32})^3 + a_2(2^{32})^2 + a_1(2^{32})^1 + a_0(2^{32})^0$$

$$A_{128-bit}(2^{32}) = a_3(2^{96}) + a_2(2^{64}) + a_1(2^{32}) + a_0$$

One of the special characteristics of this special prime, *P* (known as Solinas Prime) can be used to overcome this issue. This prime number, *P* selected is equal to $2^{64} - 2^{32} + 1$. Using this prime to generate the finite field has the following properties:

1) $2^{192}\ mod\ P = 1$
2) $2^{160}\ mod\ P = 1 - 2^{32}$
3) $2^{128}\ mod\ P = -2^{32}$
4) $2^{96}\ mod\ P = -1$
5) $2^{64}\ mod\ P = 2^{32} - 1$

By substituting the fourth and fifth into the previous equation yields:

$$A_{128-bit}(2^{32})\ mod\ P = (2^{32})(a_2 + a_1) - (a_3 + a_2) + a_0$$

By definition, multiplication of two double precisions numbers, **A** and **B** is equal to:

$$C = A * B$$

$$C_{128-bit} = (2^{64})(A_hB_h) + (2^{32})(A_hB_l + A_lB_h) + A_lB_l$$

Where *h* denotes the higher precision and *l* denotes the lower precision of the operands, and

$$Let\ Z = A_hB_h;$$

$$Let\ Y = (A_hB_l + A_lB_h);$$

$$Let\ \mathrm{X} = A_lB_l;$$

$$C_{128-bit} = (2^{64})(Z) + (2^{32})(Y) + X$$

The values of $a_3, a_2, a_1\ and\ a_0$ can be extracted from:

$$a_0 = X_l$$

$$a_1 = Y_l + X_h\,; ignore\ overflow\ MSB, store\ to\ a_{1\_cout};$$

$$a_2 = Z_l + Y_h + a_{1\_cout}; ignore\ overflow\ MSB, store\ to\ a_{2\_cout};$$

$$a_3 = Z_h + a_{2\_cout};$$

### 4.2 Large Integer Multiplication on NVIDIA GPU with Kepler Architecture

In this work, an SSMA function that is capable of performing (512 * 512)-bit multiplication is implemented with three 64-point NTTs (two forward NTTs and one inverse NTT). The Cooley-Tukey FFT method is used to speed up the 64-point NTT by computing a one dimensional 64-points NTT in two dimensional NTT of eight 8-point NTTs (denotes as (8 * 8)-point CTFNT) to increase parallel computability. Next, CRT is applied on the top level of the mentioned algorithm to combine multiple SSMAs of (512 * 512)-bit multiplications and achieve multiplication of larger bit size.

This multi-precision multiplication algorithm is implemented on three different GPU memory levels: 1) global memory, 2) shared memory and 3) registers, to compare and study the performance with different memory latency. The implementation on registers is the main propose idea of this project. The data stored in registers of a thread are not visible to the others before NVIDIA released GPU with Kepler architecture in 2012, which come with a new feature known as “Shuffle Instruction” that allowed the threads within the same warp to read the registers data of each other without the needs of going up to memory of higher latency.

This implementation is able to perform up to 900 multiplications of 1024-bit, 2048-bit, 4096-bit and 8192-bit multiplication simultaneously on average of 0.095ms, 1.119ms, 1.132ms and 1.113ms respectively, and achieved maximum

speedup of 11.45% and 36.54% compared to global memory and shared memory implementation.

### 4.2.1 NTT Implementation

Number Theoretic Transform (NTT) appears to be the most crucial parts in the SSMA algorithm, as a standard SSMA algorithm involves three NTTs (two Forward NTTs and one Inverse NTT) and one convolution. Convolution is a rather simple and direct part of the algorithm, since it is point-wise multiplication between both of the transformed operands in frequency domain, with a computational complexity of O($n$), where $n$ is the number of precisions. However, the computational complexity of a primitive NTT algorithm is equal to O($n^2$). Hence, the Cooley-Tukey Fast Fourier Transform (CTFFT) method typically used to speed up Discrete Fourier Transform (DFT) is used in this work to reduce the computational complexity of NTT. The use of CTFFT method on NTT (denoted as CTFNT) can efficiently reduce the computational complexity from O($n^2$) to O($n \log n$).

To use NTT in SSMA, the modulus, $M$ has to be co-prime with the NTT size, $N$, so that the $N^{th}$ primitive root of unity, $g$ exists. In this project, the 64-bit Solinas prime number, $P = 0xFFFFFFFF00000001$ is selected. Since a prime number is always co-prime with any other numbers, the requirement is fulfilled. This special prime number has the primitive root of unity in power-of-2 for several small NTT sizes, relaxing the needs of twiddle factors multiplication in NTT by using left shifting instead of exact multiplication. Secondly, the bit size of each of the elements in the NTT, $n$ has to meet the requirement:

$$(N/2)(2^n-1)^2 < P.$$

**Table 4.1 List of primitive root of unity**

| N | Primitive root of unity, $g$ | Number of bit (left shifting) |
|---|---|---|
| 4 | 0x1000000000000 | 48 |
| 8 | 0x1000000 | 24 |
| 16 | 0x1000 | 12 |
| 32 | 0x40 | 6 |
| 64 | 0x8 | 3 |
| 192 | 0x2 | 1 |
| 6 | 0x100000000 | 32 |
| 12 | 0x10000 | 16 |
| 24 | 0x100 | 8 |
| 48 | 0x10 | 4 |
| 96 | 0x4 | 2 |

Table 4.1 shows the primitive root of unity, $g$ for different values of $N$ and respective number of bit for left shifting.

### 4.2.2 SSMA Implementation

The (512 * 512)-bit SSMA is implemented with 64-point NTT, with all of the elements in NTT being set to 16-bit each. Since half of the elements have to be reserved for zero-padding, the operand size for the multiplication is equal to 32 points * 16-bit = 512-bit. The 64-point NTT can be decomposed into two dimensional CTFNT of (1 * 64), (2 * 32), (4 * 16), (8 * 8), (16 * 4), (32 * 2), or (64 * 1) points. The experiments for all of the possible combinations mentioned above has been carried out and the results shown that the (8 * 8)-point CTFNT combination has the most optimum performance in term of computational time.

Figure 4.1 illustrates the flow of the SSMA function. Both of the operands, the multiplicand, X and the multiplier, Y are randomly generated using the GMP library. Firstly, both of these operands are broken into their multi-precisions representation of 64 limbs of 16-bit each. Half of the most significant limbs are filled with zeroes before forward transformed into frequency domain. Secondly, convolution is carried out to get the product, Z in frequency domain. Thirdly, Z is inverse transformed back to its time domain. Lastly, evaluation is performed to get the exact product. Note that the small letter x, y and z with subscripts (e.g. $x_0$) are used to denote the data in time domain while capital letter X, Y, and Z with subscripts are used to represent the data in frequency domain. The arrow bars on the right show the flow of the program from start to finish and the execution running in either CPU or GPU.

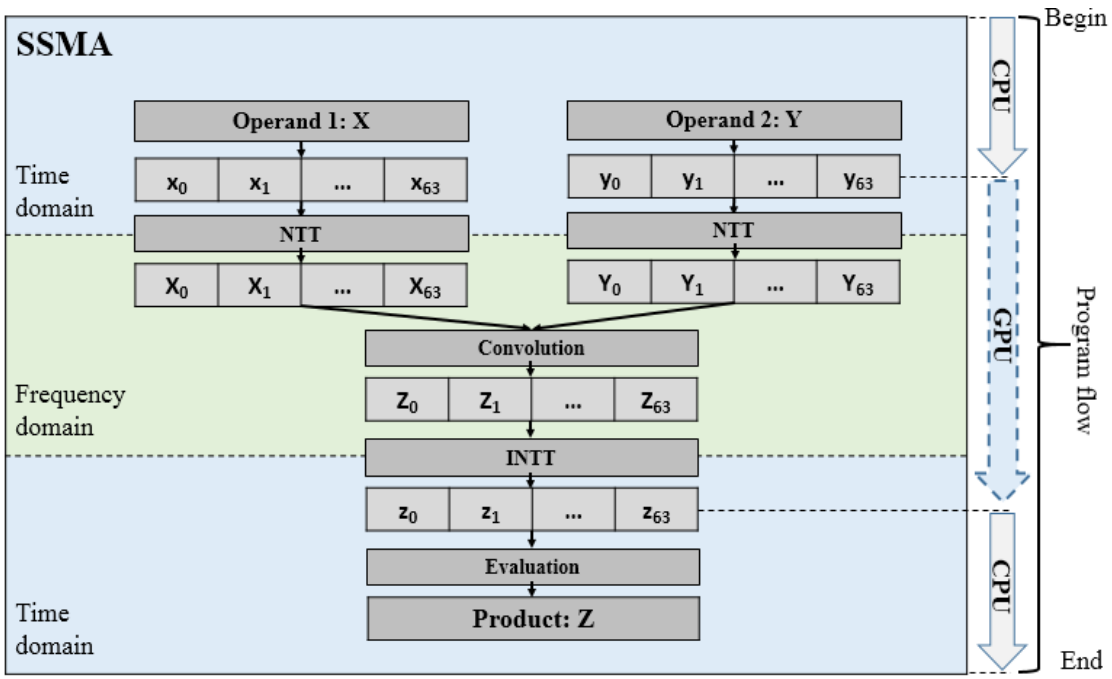


**Figure 4.1 SSMA flow illustration**

In GPU kernel, each SSMA is set to run in one block of two dimensional threads. The *x*-dimension is fixed with eight threads, whereby each of this thread is able to compute a single 8-point NTT independently, together a 64-point NTT ((8 * 8)-point CTFNT) can be computed. The *y*-dimension is used to determine the number of CRT moduli, *c*, for the large integer multiplication. The value of *c* is configurable between 4, 8, 16 or 32 for operand size of 1024-bit, 2048-bit, 4096-bit or 8192-bit respectively. The number of CRT moduli, *c*, are corresponds to the number of SSMA run in the program.

Figure 4.2 illustrates how eight threads are launched to compute a single SSMA and Figure 4.3 shows how *c* rows of these eight threads can be replicated in *y*-dimension for multiple SSMAs according to number of CRT moduli.

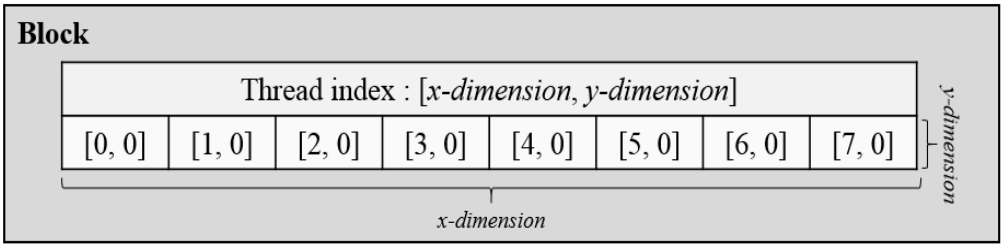


**Figure 4.2 Eight threads in one block**

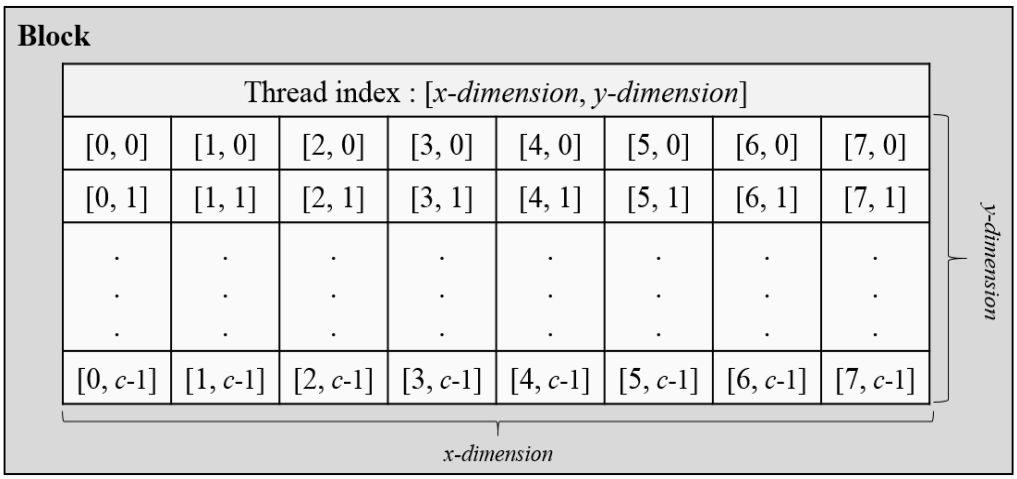


**Figure 4.3 Two-dimensional threads in one block**

### 4.2.3 CRT Implementation

The bottleneck of the SSMA implementation discussed in the previous section is the maximum operand size supported, which is limited to only 512-bit. To achieve large integer multiplication with larger operand size, the idea of concatenating multiple SSMAs to support larger operand size can be done by implementing CRT on top of these SSMAs.

Figure 4.4 illustrates the implementation of CRT algorithm on top of several SSMAs. The implementation in this work supports 1024-bit, 2048-bit, 4096-bit and 8192-bit multiplication, the size of the input operands shown in Figure 4.2 with configuration of $c = 4$, $c = 8$, $c = 16$, and $c = 32$ respectively, where $c$ is the number of CRT moduli required for the CRT and ICRT operations. The number of CRT moduli, $c$, can be determined by dividing the bit size of the product (sum of the bit size of both the input operands) with 512, the bit size of one single CRT moduli.

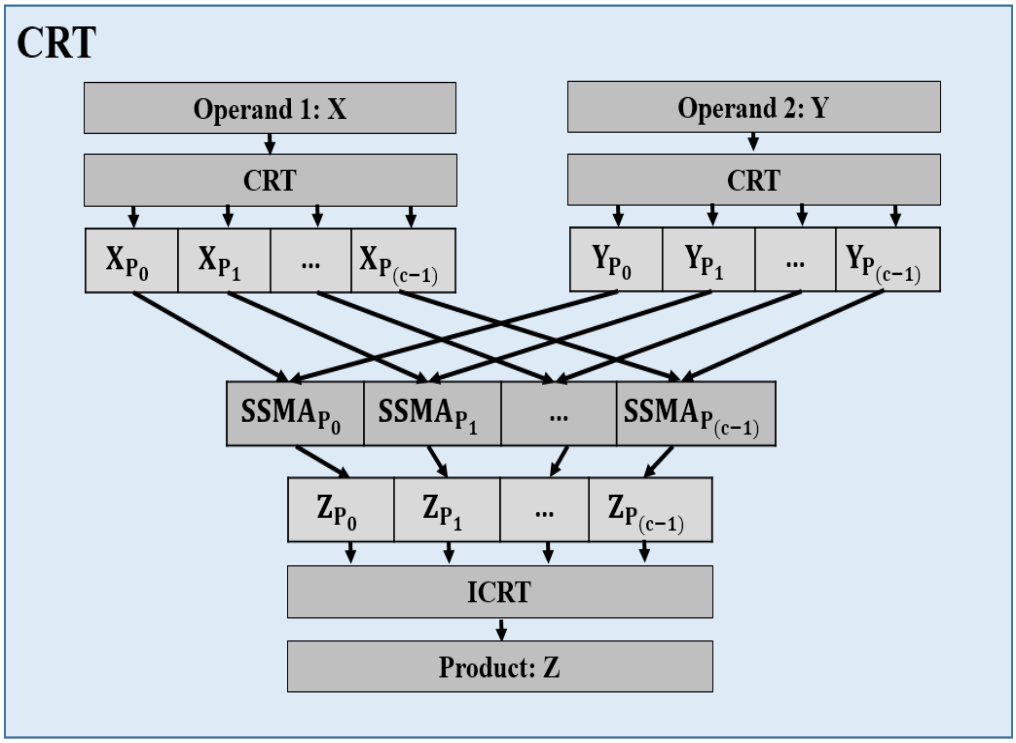


**Figure 4.4 CRT implementation illustration**

From Figure 4.4, two of the input operands, **X** and **Y** are first converted into two series of CRT coefficients, $X_{P_0}$ to $X_{P_{(c-1)}}$ and $Y_{P_0}$ to $Y_{P_{(c-1)}}$ by performing forward CRTs, which divide **X** and **Y** with *c* numbers of 512-bit CRT moduli to get the residue numbers. Both the large integer operands and the set of 512-bit CRT moduli are generated randomly using GMP library. Two series of these **X** and **Y** CRT coefficients are then paired according to their indices and fed into multiple SSMAs simultaneously to produce the series of CRT coefficients of the product, $Z_{P_0}$ to $Z_{P_{(c-1)}}$. Lastly, Inverse CRT is performed on this series of coefficients to get the final product, **Z**.

However, the implementation discussed as in Figure 4.4 is not able to support exact x-bit multiplication (x = 1024, 2048, 4096, 8192). This is because the bit size product of all the CRT moduli $(\boldsymbol{P} = \prod_{i=0}^{c-1} P_i)$, is slightly less than the expected product size. For example, in (8192 * 8192)-bit multiplication, $c$ = 32, the bit size of **P** is equal to 16372-bit instead of the expected 16384-bit. This problem has causes the maximum operand sizes supported implementation to be 1023-bit, 2046-bit, 4091-bit and 8185-bit instead of expected 1024-bit, 2048-bit, 4096-bit and 8192-bit respectively.

To overcome this problem, an extra CRT modulus can be added to increase the bit size of **P**, and hence able to compensate the missing bits of the expected operand size. Adding an extra 512-bit can be costly in terms of computation time. Hence, a small modulus of 16-bit is used instead. Figure 4.5 shows the flow of the implementation after adding in the extra CRT modulus.

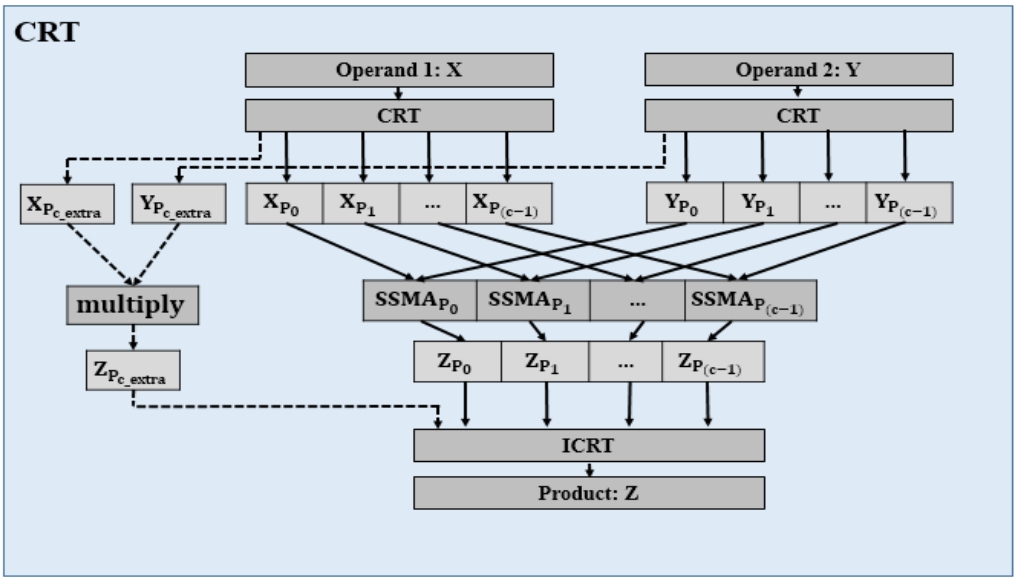


**Figure 4.5 CRT implementation with extra modulus illustration**

### 4.2.4 Memory allocation

The twiddle factors are needed for the CTFNT are precomputed before passing into the GPU memory to save computation time. The first version of the implementation stores the precomputed twiddle factors in the global memory of the GPU for the ease of access and sharing between threads from different blocks. The twiddle factors stored are being read frequently as they are needed in every iteration of the CTFNT. Hence, in the second version of the implementation, the shared memory of the GPU is used to store the precomputed twiddle factors to improve the performance by reducing the memory read latency.

However, this implementation introduced bank conflict when reading data from the shared memory. Bank conflict happened when more than one thread are trying to read data of different locations from the same memory bank. In this case, the memory read operations from the shared memory will be serialized. Bank conflict happened due to the memory access pattern of the CTFNT, which access the shared memory in a column-by-column manner followed by row-by-row manner in its column-NTTs and row-NTTs operations respectively. During the column-by-column memory access, each of the threads are reading the data from different memory banks, no bank conflict occurred. When the threads are reading the data in row-by-row manner, all of the threads are trying to read the data from the same memory bank, bank conflict happened.

Figure 4.6 and Figure 4.7 illustrate the memory access pattern in column-by-column and row-by-row manners, from the first access to eight access across

eight different banks. The numbers 0 to 63 illustrate the indices of the twiddle factors stored.

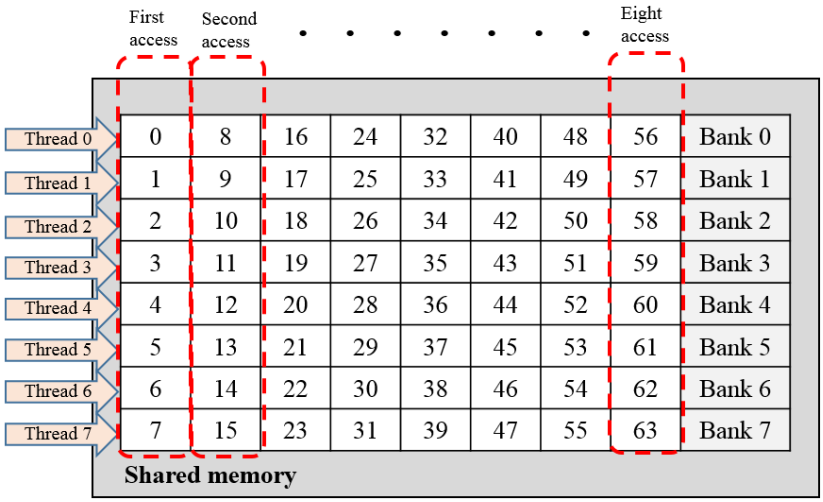


**Figure 4.6 Column-by-column access: No bank conflict**

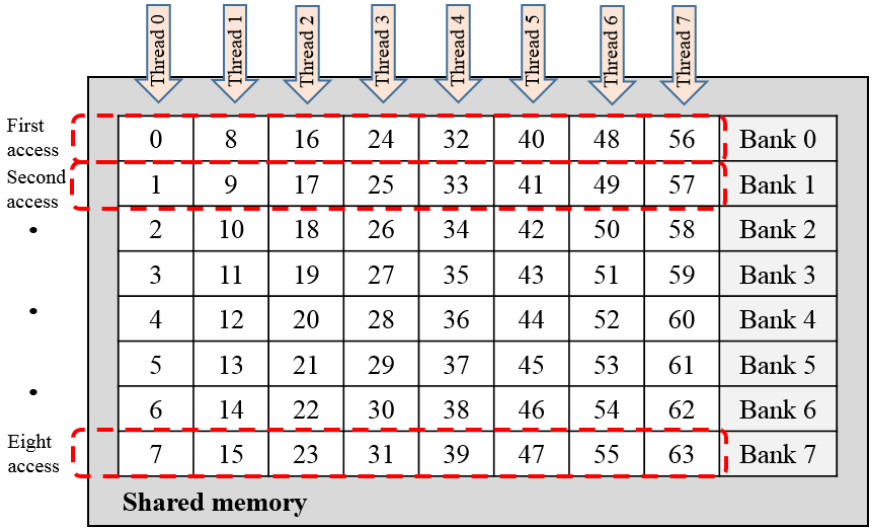


**Figure 4.7 Row-by-row access: Bank conflict**

In GPU memory hierarchy, the GPU registers are the memory with lowest latency. Conventionally, the data stored inside registers of a thread are only limited for the uses of the thread itself, which is not visible to the others. In 2012, NVIDIA has released new generation of GPU, the GPU with Kepler architecture. GPU from this generation and onwards come with a new feature known as "Shuffle Instruction" that allowed the threads within the same warp of the GPU to read the registers data from each other without the needs of going through shared memory or global memory.

In our third version of the implementation, we proposed a novel method by utilizing the "Shuffle Instruction" to store the precomputed twiddle factors into the registers, reducing the memory latency from both global memory and shared memory implementation and bank conflict in shared memory can be avoided. In this implementation, two sets of precomputed twiddle factors, the set of twiddle factors for 64-point NTT and 8-point NTT (for the uses of CTFNT) are distributed among the threads within the same warp, each thread will hold four twiddle factors, two from each set.

During the computation, all of the threads will shuffle their registers to the respective thread that holding the twiddle factors with the corresponding index, this process is repeated in each iteration until all of the NTTs are computed.

### 4.3 Large Integer Multiplication on NVIDIA GPU with Pascal Architecture

This implementation is an improvement over our previous work (Section 4.2) in term of multiplication operands size supported. The bottlenecks of our previous large integer multiplication algorithm implementation are the CRT and ICRT operations, which is needed to combine multiple (512-bit * 512-bit) SSMAs to achieve multiplication with larger bit size.

In this implementation, we are able to eliminate CRT in replace of extra levels of CTFNT. This approach increased the NTT size (number of points supported in an NTT) to accommodate more precisions for operand of larger size, and using only one SSMA is sufficient to compute one large integer multiplication. However, the advantage of using left shifting for twiddle factors multiplication is not available for NTT of larger size as the values are not a power of two within the finite field.

**Table 4.2 Primitive root of unity for N = 4096, 16384, 32768 and 65536**

| N | Primitive root of unity, $g$ |
|---|---|
| 4096 | 0x984E_C519_4D00_5735 |
| 16384 | 0x97B0_081F_0175_31DF |
| 32768 | 0x2E43_53DF_CE41_DEAF |
| 65536 | 0xDC92_18A8_6D10_F3A3 |

Table 4.2 shows the primitive root of unity, $g$, for NTT size of 4096, 16384, 32768 and 65536 in hexadecimal form. The modular multiplier discussed in section 4.1.3 is used for the multiplication instead.

NTTs with NTT size as listed in Table 4.2 are implemented with multi-level CTFNT. This work is implemented on NVIDIA GPU GTX1070 with Pascal architecture released on year 2016. Our work is able to achieve multiplication with operands size of 49,152-bit, 196,608-bit, 393,216-bit and 786,432-bit in less than 1ms, 1.12ms, 1.24ms and 1.37ms respectively.

### 4.3.1 CTFNT Implementation

#### 4.3.1.1 CTFNT Decomposition

Generally, Cooley-Tukey Fast Number Theoretic Transform (CTFNT) divides an $N$-point NTT into ($N_1$ * $N_2$)-point NTT, where $N = N_1 * N_2$. One level of CTFNT involves three steps: the column NTTs, followed by the twiddle factors multiplication for $N$ and end with the row NTTs. From the CTFNT equation in section 3.6.4, the inner summation is known as the column NTTs, while the outer summation is the row NTTs. Each of these $N_1$-points column NTTs and $N_2$-points row NTTs can be further divided by applying another level of CTFNT.

In this work, 4096-point NTT is implemented with (64 * 64)-point CTFNT, where each of the 64-point column NTTs are implemented with (8 * 8)-point CTFNT *(4096-point NTT = (64 * (8 * 8))-point CTNTT)*. We set this 4096-point NTT as base case by fixing $N_1$ = 4096 and configure only $N_2$ at the values of 4, 8 and 12 to compute NTT for NTT size of 16384, 32768 and 65536 respectively.

**Table 4.3 CTNTT decomposition and multiplication size supported**

| | NTTSIZE, $N$ | $N_1$ | $N_2$ | CTFNT decomposition | | Operand size (bit) | Product size (bit) |
|---|---|---|---|---|---|---|---|
| 1 | 4096 | 4096 | 1 | 1 * (64 * (8 * 8)) | | 49,152 | 96K |
| 2 | 16384 | 4096 | 4 | 4 * (64 * (8 * 8)) | | 196,608 | 384K |
| 3 | 32768 | 4096 | 8 | 8 * (64 * (8 * 8)) | | 393,216 | 768K |
| 4 | 65536 | 4096 | 16 | 16 * (64 * (8 * 8)) | | 786,432 | 1,536K |

Table 4.3 shows the size of multiplication operands supported and the level of CTFNT decomposition of our implementation. All four of the NTTs are implemented with three levels of CTFNT.

- First level : $N = N_1 * N_2$
- Second level : $N_1 = N_{11} * N_{12} \rightarrow N = (N_{11} * N_{12}) * N_2$
- Third level : $N_{11} = N_{111} * N_{112} \rightarrow N = ((N_{111} * N_{112}) * N_{12}) * N_2$

In our implementation, the second and third level of the CTFNT decomposition are done symmetrically, where the number of row NTTs is equal to the number of column NTTs (*e.g.: Let* $N_1 = N_2$*, hence* $N = N_1^2$ *or* $N = N_2^2$). Only one set of precomputed twiddle factors is needed to store into GPU as they are the same for both of the CTFNT's columns NTTs and row NTTs, hence reduced the memory required.

#### 4.3.1.2 CTFNT Kernels Implementation

Three GPU kernels are designed to compute CTFNT for both the multiplication operands simultaneously. Each of these kernels in charge of handling each of the three steps of the first level CTNTT:

a) Kernel 1: $N_2$ number of column NTTs ($N_2$ * $N_1$-point NTTs)

b) Kernel 2: Twiddle Factors Multiplication ($N$ point-wise multiplication)

c) Kernel 3: $N_1$ number of row NTTs ($N_1$ * $N_2$-point NTTs)

In the first kernel, a block of 64 threads are designed to compute the base case, 4096-point NTT by dividing it into (64 * 64)-point CTFNT, where each of these threads compute a single 64-point NTT independently. These 64-point NTTs are further divided into (8 * 8)-point CTFNT within the threads themselves. A total number of (2 * $N_2$) of these blocks are launched in this kernel to compute the column NTTs of the first level CTFNT in our implementation, where the lower $N_2$ blocks (block 0 to block $N_2 - 1$) compute forward column NTTs for operand X while upper $N_2$ blocks (block $N_2$ to block 2 * $N_2 - 1$) compute forward column NTTs for operands Y. In short, the first kernel is used to compute $N_2$ number of 4096-point column NTTs for both of the multiplication operands. Figure 4.8 illustrates the blocks and threads allocation in first kernel.

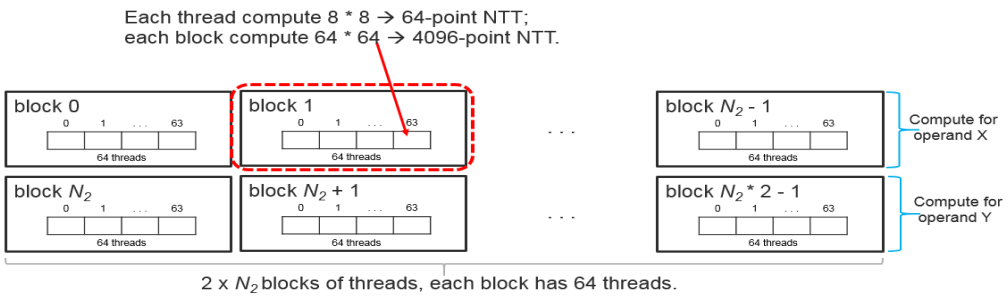


**Figure 4.8 CTFNT First Kernel**

The second kernel is used for twiddle factors multiplication, which perform one to one multiplication of the transformed data from the first kernel with the twiddle factors of their respective indices. *N* number of threads are spawned where eat of the threads will perform one multiplication for operand X and one multiplication for operand Y. Since the maximum number of threads can be spawned in a single block is limited to 1024, *N*/1024 number of blocks are launched to have *N* number of threads. Figure 4.9 illustrates the blocks launched and threads distribution.

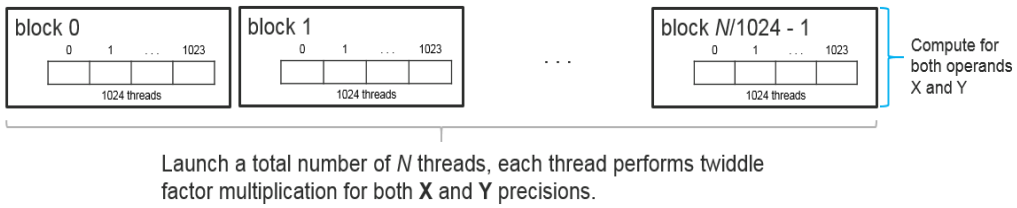


**Figure 4.9 CTFNT Second Kernel**

The third kernel is used to compute $N_1$ number of row NTTs, which is equal to 4096 $N_2$ point-NTTs as the value of $N_1$ is fixed at the value of 4096 in our implementation. In this kernel, we launch 8 blocks with 1024 threads each, where block 0 to block 3 is the lower set of block and block 4 to block 7 is the upper set of block. Hence, both sets of block will have a total of 4096 threads each. The lower set of block computes row NTTs for operand X whereas the upper set of block computes row NTTs for operand Y. Figure 4.10 shows the blocks and threads distribution for the third kernel.

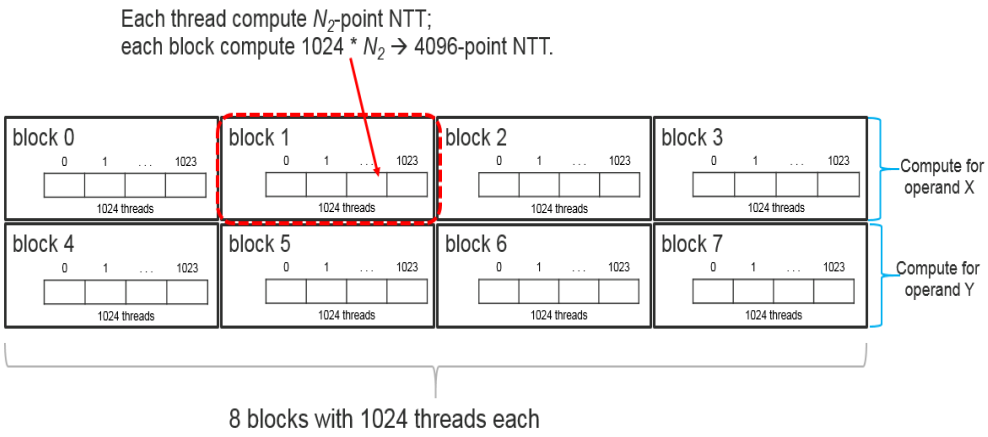


**Figure 4.10 CTFNT Third Kernel**

Each of the blocks is used to compute 1024 $N_2$-point row NTTs, where each of the threads within the block is in charge of computing a single $N_2$-point row NTTs. Hence, the lower or upper set alone is capable of computing 4096 $N_2$-point NTTs *(4 blocks * 1024 threads * $N_2$ –point NTT = 4096 $N_2$ point-NTTs).*

For NTT size equal to 4096, only kernel one is needed to complete the transformation.

### 4.3.2 SSMA Implementation

The SSMA in this work utilized the CTFNT implemented in section 4.3.1.1 to perform the forward transform for both the input operands. Next, the transformed operands are sent to another GPU kernel to perform convolution (point-wise-multiplication of the operands in frequency domain). This convolution kernel implementation is similar to the kernel 2 of CTNTT implementation as discussed in section 4.3.1.2, except that the inputs are both the operands instead of the operands with the precomputed twiddle factors. The CTFNT from the previous section is used for the inverse CTFNT part of the SSMA with slightly modifications:

1) Only half of the blocks are launched as there is only one operand Only half of the blocks are launched as there is only one operand (the convolution product, Z) has perform inverse transformation;
2) The twiddle factors of inverse transform are used instead of forward transform;
3) The forth kernel, kernel 4 is implemented for the modular multiplicative inverse $N^{-1}$ multiplication needed for inverse transform;

The kernel 4 is similar to the convolution kernel in this SSMA implementation, except that the first input is the product, Z while the second input of the function is the modular multiplicative inverse, $N^{-1}$ instead of a series of operand precisions. The algorithm flow of this SSMA implementation is same as the flow illustrated in Figure 4.1 with $N$ of larger size. The implementation is illustrated in Figure 4.11 on the following page.

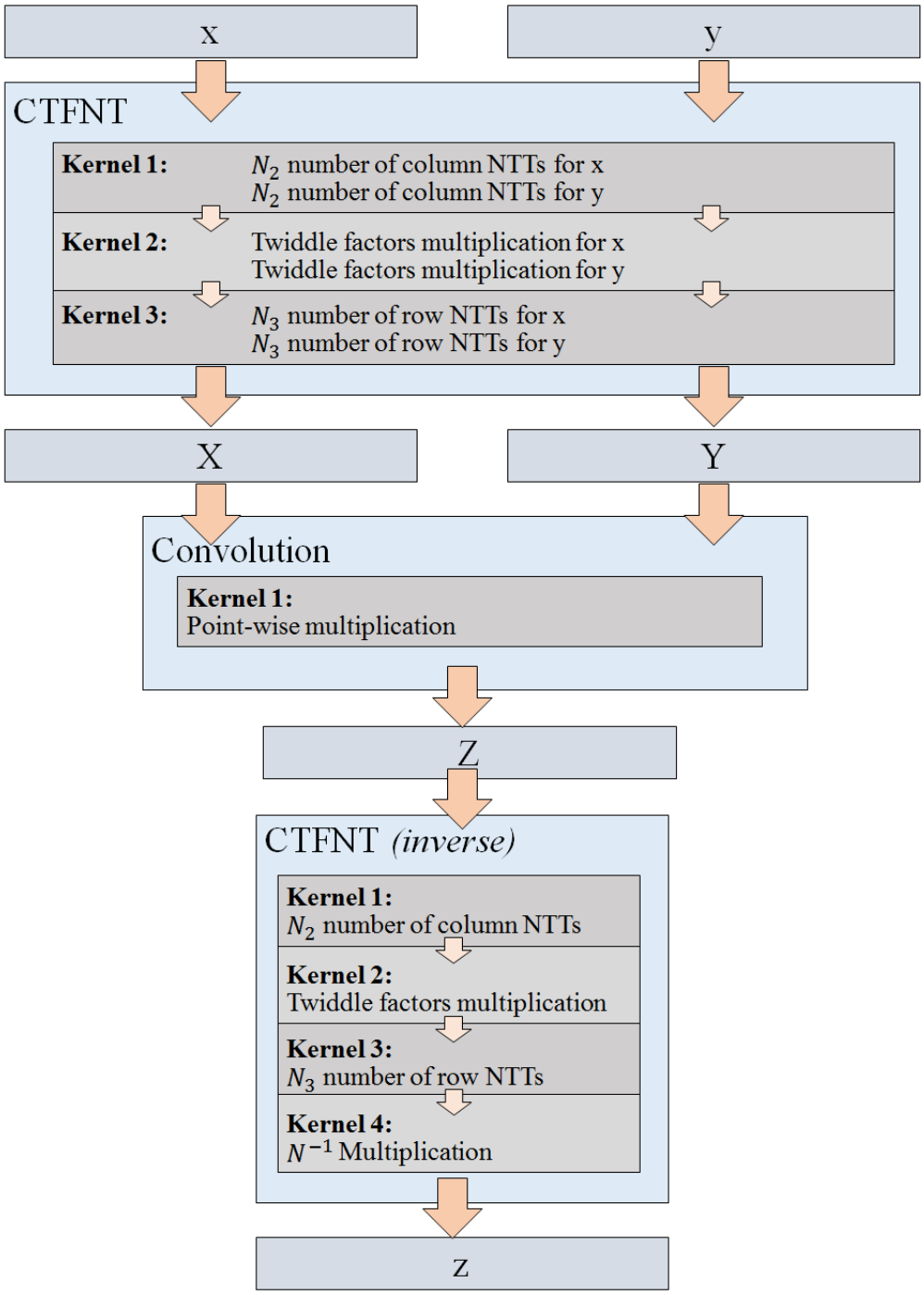


**Figure 4.11 SSMA kernels implementation**

### 4.4 FPGA Implementation

A hardware module that is capable of computing one level of (4 * 4)-point CTFNT with multiplication is designed in this work. This module utilized BRAM to store the intermediate data repeatedly to compute a 256-point NTT. Extra control circuits are added into the implementation to perform SSMA by utilizing the developed 256-point NTT module for three NTTs (two forward and one inverse) and convolution. This SSMA design is able to compute multiplication of 3072-bit.

The hardware modules in this work are designed using Verilog HDL. The functional and timing behavior verification of the designed hardware are done in Vivado HLx edition 2016.3 EDA tool. The clock speed is configured at 20MHz the targeted FPGA board: Nexys4 DDR board with Xilinx Artix-7 FPGA (part number = xc7a100tcsg324-1).

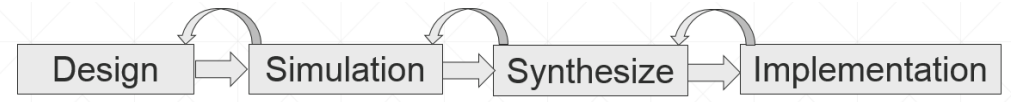


**Figure 4.12 FPGA development cycle**

Figure 4.12 shows the development cycle of a FPGA design used in our implementation. Starting with designing the hardware modules in Verilog HDL, followed by simulation to verify the functional behavior of the design using Modelsim. In this phase, the number of clock cycles required to process the

intended operation will be shown regardless of the clock frequency. Next, the verified design is ported and synthesized in Xilinx Vivado. This is to ensure that the design can be developed into an actual hardware and an estimation of hardware resources utilization on a targeted FPGA board will be shown. Lastly, the synthesized design is implemented to the target board during the implementation phase based on the design constraints. This is the most critical phase throughout the development as the implementation result will show whether the synthesized design can meet the timing requirement decided by the developer. The implementation result will also provide the developer the power consumption of the design. The development will go back to the previous phase for optimization or when any error(s) happened in either one of the phases.

Section 4.4.1, 4.4.2 and 4.4.3 describe three different design and development phases of this implementation.

### 4.4.1 Preliminary Design: SSMA with typical NTT

Table 4.4 shows an example of how each of the output of an 8-point NTT can be computed using the forward transform NTT equation from section 3.6.3. Table 4.5 shows a simplified version of Table 4.4 with the addition operators are omitted for better readability. The indices below indicate the index of the respective input data.

**Table 4.4 Computation Equation of an 8-point NTT**

| Output, $X_j$ | Computation equation |
|---|---|
| $X_0$ | $x_0g^0 + x_1g^0 + x_2g^0 + x_3g^0 + x_4g^0 + x_5g^0 + x_6g^0 + x_7g^0$ |
| $X_1$ | $x_0g^0 + x_1g^1 + x_2g^2 + x_3g^3 + x_4g^4 + x_5g^5 + x_6g^6 + x_7g^7$ |
| $X_2$ | $x_0g^0 + x_1g^2 + x_2g^4 + x_3g^6 + x_4g^8 + x_5g^{10} + x_6g^{12} + x_7g^{14}$ |
| $X_3$ | $x_0g^0 + x_1g^3 + x_2g^6 + x_3g^9 + x_4g^{12} + x_5g^{15} + x_6g^{18} + x_7g^{21}$ |
| $X_4$ | $x_0g^0 + x_1g^4 + x_2g^8 + x_3g^{12} + x_4g^{16} + x_5g^{20} + x_6g^{24} + x_7g^{28}$ |
| $X_5$ | $x_0g^0 + x_1g^5 + x_2g^{10} + x_3g^{15} + x_4g^{20} + x_5g^{25} + x_6g^{30} + x_7g^{35}$ |
| $X_6$ | $x_0g^0 + x_1g^6 + x_2g^{12} + x_3g^{18} + x_4g^{24} + x_5g^{30} + x_6g^{36} + x_7g^{42}$ |
| $X_7$ | $x_0g^0 + x_1g^7 + x_2g^{14} + x_3g^{21} + x_4g^{28} + x_5g^{35} + x_6g^{42} + x_7g^{49}$ |

**Table 4.5 Computation Equation of an 8-point NTT (Simplified)**

| Output, $X_j$ | Computation equation | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $X_0$ | $x_0g^0$ | $x_1g^0$ | $x_2g^0$ | $x_3g^0$ | $x_4g^0$ | $x_5g^0$ | $x_6g^0$ | $x_7g^0$ |
| $X_1$ | $x_0g^0$ | $x_1g^1$ | $x_2g^2$ | $x_3g^3$ | $x_4g^4$ | $x_5g^5$ | $x_6g^6$ | $x_7g^7$ |
| $X_2$ | $x_0g^0$ | $x_1g^2$ | $x_2g^4$ | $x_3g^6$ | $x_4g^8$ | $x_5g^{10}$ | $x_6g^{12}$ | $x_7g^{14}$ |
| $X_3$ | $x_0g^0$ | $x_1g^3$ | $x_2g^6$ | $x_3g^9$ | $x_4g^{12}$ | $x_5g^{15}$ | $x_6g^{18}$ | $x_7g^{21}$ |
| $X_4$ | $x_0g^0$ | $x_1g^4$ | $x_2g^8$ | $x_3g^{12}$ | $x_4g^{16}$ | $x_5g^{20}$ | $x_6g^{24}$ | $x_7g^{28}$ |
| $X_5$ | $x_0g^0$ | $x_1g^5$ | $x_2g^{10}$ | $x_3g^{15}$ | $x_4g^{20}$ | $x_5g^{25}$ | $x_6g^{30}$ | $x_7g^{35}$ |
| $X_6$ | $x_0g^0$ | $x_1g^6$ | $x_2g^{12}$ | $x_3g^{18}$ | $x_4g^{24}$ | $x_5g^{30}$ | $x_6g^{36}$ | $x_7g^{42}$ |
| $X_7$ | $x_0g^0$ | $x_1g^7$ | $x_2g^{14}$ | $x_3g^{21}$ | $x_4g^{28}$ | $x_5g^{35}$ | $x_6g^{42}$ | $x_7g^{49}$ |
| Index, *i* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

From Table 4.5, if only the set of input data, $x_i$ is being read from the computation equation section vertically column-by-column, it is clear that they are the same for the set of output data, $X_j$. Likewise, if only the set of twiddle factors, $g^{ij}$ is being read from the computation equation section vertically

column-by-column, we notice that the power for the set of twiddle factors are actually increased by the respective indices of the set of input data as *j* increased. These two facts indicate that all of the products of input with their respective twiddle factors can be reused to compute the subsequence output after computing current output by multiplying the products with $g^{i}$ instead of $g^{ij}$ except for the first output, $X_0$ where the multiplication with $g^0$ are redundant and can be neglected. For instance, after computing the summation of products for $X_1$, the set of product ($x_0g^0$, $x_1g^1$, …, $x_7g^7$) can be fed right back to the circuit, perform multiplication again with the set of twiddle factors ($g^1$, $g^2$, …, $g^7$) and compute the summation of product for $X_2$.

Figure 4.13 shows the block diagram of the designed 8-point NTT processing unit. From Figure 4.13, ip_x0, ip_x1, …, ip_x7 are eight of the input of the 8-point NTT. This set of input is fed into an adder to compute the sum of product, which is the output data. At the same time, each of these input data are fed into a shifter to perform twiddle factors multiplication. The set of output from these shifters ($p_0$, $p_1$, …, $p_7$) are fed into the next 8-point NTT processing unit to compute the next output. Figure 4.14 shows how eight of these 8-point NTT processing units can be instantiated to compute all eight output data.

**Initial Design**

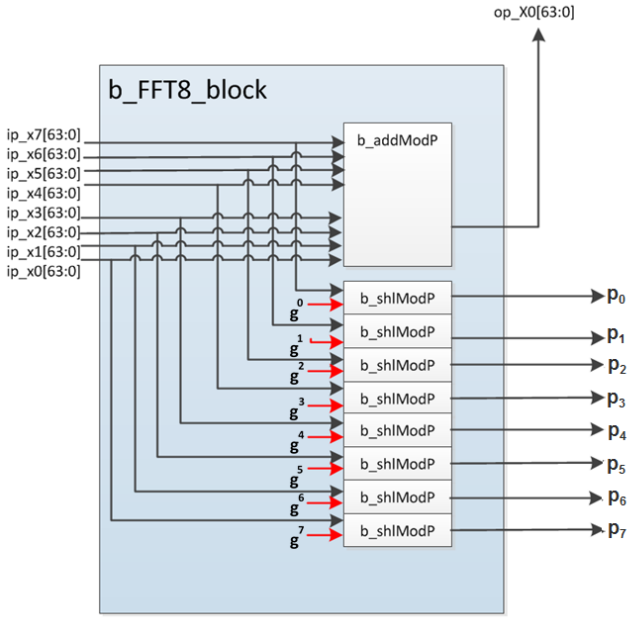


**Figure 4.13 Block diagram of 8-point NTT processing unit**

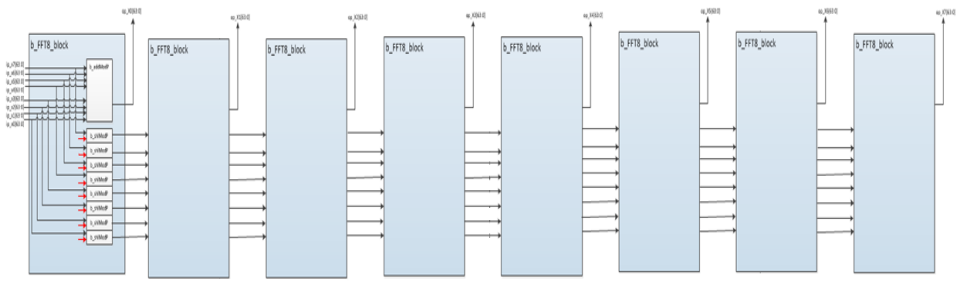

**Figure 4.14 Block diagram of full 8-point NTT processing unit**

Ideally, by instantiating and concatenating eight of the design as shown in Figure 4.14, we are able to compute all of the eight point NTT output in one clock cycle. However, this implementation is impractical as the output from each of the individual module are dependent on the output of the previous module. This creates a very long critical path (from the input of the first module to the last

NTT output). For instance, the latency of one individual module (from module input to module output) is 50ns when synthesized. Thus, the last NTT output takes 400ns to compute (minimum clock frequency required = 2.5MHz). This will also cause all the previous modules to be idle once they have computed their respective output as the next set of 8-point NTT input data have to queue until the last output data of current 8-point NTT is computed. This architecture design costs a lot of hardware resources, but only achieve low occupancy, hence it is an inefficient design. Thus, the design in Figure 4.14 is then modified with pipelining technique implementation. This implementation technique allowed a single 8-point NTT processing unit to be reused for all eight points NTT, reduced the required hardware resources and supported higher clock frequency of 20MHz, which is equal to 50ns period (the latency of a single designed module).

**Improved Pipelining Design**

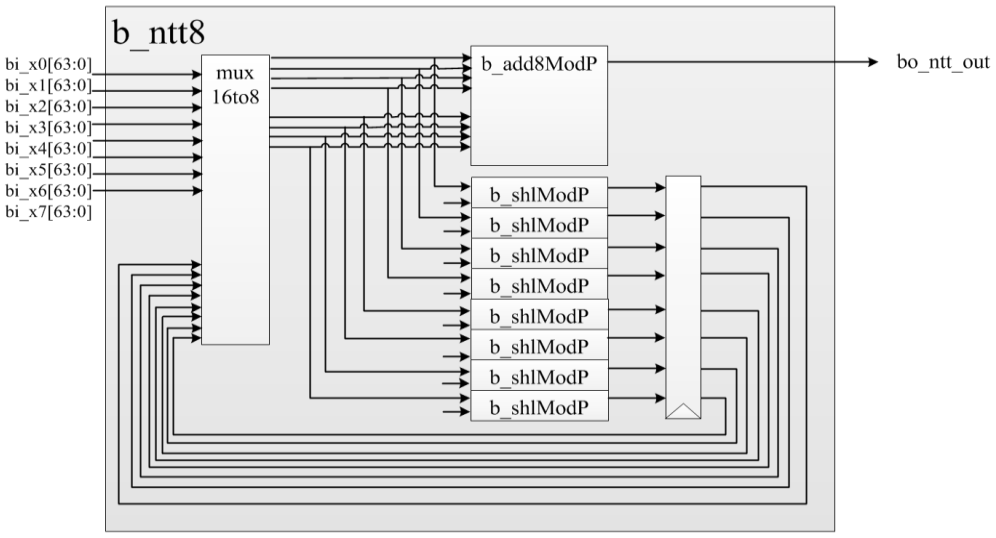


**Figure 4.15 Block diagram of pipelining 8-point NTT processing unit**

The modified design with pipelining implementation as shown in Figure 4.15 is similar to the original design as shown in Figure 4.13, except that a set of registers are used to store the products for next output and an extra 16-to-8 multiplexer is used to select the input data either from new set of input data or feedback data from previous clock cycle. Although this design took eight clock cycle to finish the computation of an 8-point NTT compared to one clock cycle of the original design, this design supports higher clock frequency. The latency of this design to complete an 8-point NTT is equal to 50ns * 8 clock cycle = 400ns, same with the original design but about 8 times lesser resources are required.

The maximum operand size supported for SSMA with 8-point NTT is equal to (8 / 2) x 24-bit = 96-bit, which is insufficient for cryptosystem. Hence, we further

modify the design as shown in Figure 4.15 to support 64-point NTT by increasing the number of input data and shifters to 64 (maximum operand size: (64 / 2) x 24-bit = 768-bit, sufficient for ECC). However, the synthesized result shows an estimated resources utilization of 373.59%. The latency of the combination circuits in the 64-input adder also increased drastically.

To overcome the problems of resources overutilization and latency, the design as shown in Figure 4.15 is reduced to compute four data at each clock cycle. A 64-point NTT is being broken down into sixteen parts of four input data each. At each clock cycle, four input data are processed and hence 16 clock cycles are needed to process all 64 data for one output. An extra accumulator is added into the design for the summation of these sixteens parts data. An internal block RAM is also instantiated in the design to store the intermediate data. Overall, 64 output required 64/2 x 16 clock cycles = 512 clock cycles (divided by half as half of the input data are zeroes, summation with them can be neglected). The block diagram of the design is shown in Figure 4.16.

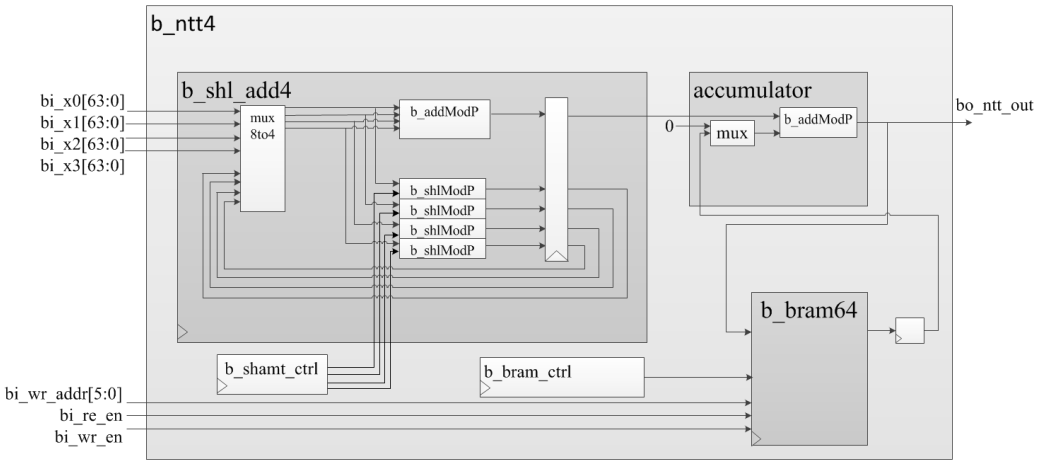


**Figure 4.16 Block diagram: 64-point NTT processing unit**

Two 64-point NTT processing units as shown in Figure 4.16 are instantiated for the SSMA module. During forward transform mode, one of the processing units is used to compute NTT for operand X and the other one is used to compute NTT for operand Y; during inverse transform mode, both of the processing units are used to compute INTT for the product, Z

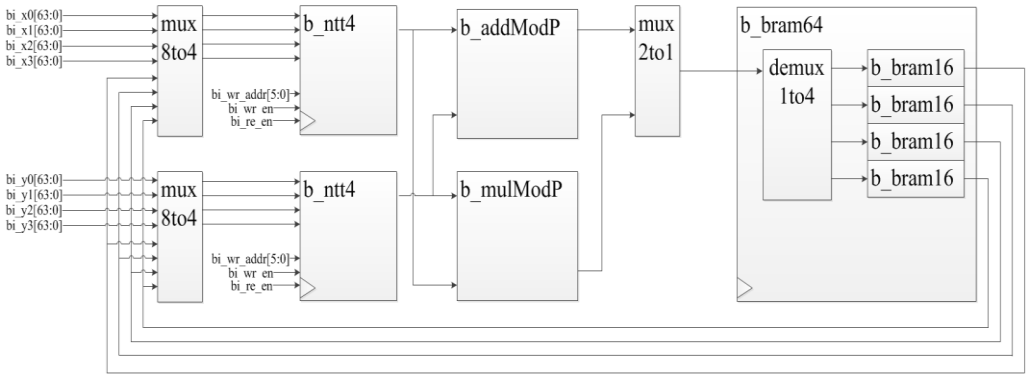


**Figure 4.17 Block diagram: SSMA module**

. Figure 4.17 shown the block diagram of the SSMA module. From Figure 4.17, the adder with the input connected to the output of both of the NTT processing units are used to compute addition of two intermediate data during inverse NTT of product Z while the multiplier with the input connected to the output of both of the NTT processing units are used for convolution of the SSMA. The 2-to-1 multiplexer follows right after the adder and multiplier are used to select the data to be stored into the block RAM. The whole SSMA took 1024 clock cycles or 51.2ms latency at 20MHz to complete (The first 512 clock cycles used for two forward NTTs running in parallel while the last 512 clock cycles are used to compute inverse NTT and the clock cycles required for convolution are hidden within the 1024 clock cycles due to pipelining).

#### 4.4.1.1 Findings

This implementation is based on a typical NTT with O($N^2$) computational complexity. It shows that how an ideal functional behavior simulation of one clock cycle 8-point NTT design can become be very different compared to the timing behavior simulation during the synthesize phase. Having only one NTT output produced at each clock cycle is highly inefficient. Hence, we modified the design from parallel-in-serial-out design discussed in this section to parallel-in-parallel-out design in next section.

### 4.4.2 Second Design: SSMA with radix-4 FNT

The goal in this design as compare to our previous preliminary design (Section 4.4.1) is to increase the supported operand size from 768-bit to 3072-bit. In this implementation, we first compare the level of decomposition of radix-*R* FNT for *R* = 2, 4, 8 and 16, with NTT size, *N* ranged from 2 to 1024 provided *N* is a power-of-2 as shown in Table 4.6. Note that a fully radix-*R* FNT is only applicable for *N* equals to the power of *R*, so some of the values in the Table is not available for radix 4, 8 and 16. Figure 4.18 illustrates the number of decomposition of radix-*R* FNT for different NTT sizes.

**Table 4.6 Level of decomposition of radix-*R* FNT against NTTSIZE, N**

| | Level of decomposition, $log_R N$ | | | |
|---|---|---|---|---|
| NTT size, *N* | Radix-2 | Radix-4 | Radix-8 | Radix-16 |
| 2 | 1 | - | - | - |
| 4 | 2 | 1 | - | - |
| 8 | 3 | - | 1 | - |
| 16 | 4 | 2 | - | 1 |
| 32 | 5 | - | - | - |
| 64 | 6 | 3 | 2 | - |
| 128 | 7 | - | - | - |
| 256 | 8 | 4 | - | 2 |
| 512 | 9 | - | 3 | - |
| 1024 | 10 | 5 | - | - |

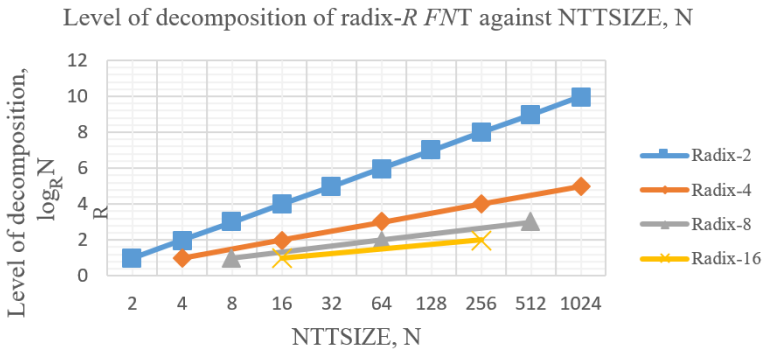


**Figure 4.18 Level of decomposition of radix-*R* FNT against NTTSIZE, N**

Table 4.6 and Figure 4.18 show that every time when the radix, *R* is doubled, the level of decomposition is reduced by half. Algorithmically, a radix-*R* FNT processes *R* number of data at a time. Hence, radix-*R* FNT with higher value of *R* is able to compute an *N*-point NTT faster. However, increasing the value of *R* will reduce the degree of parallelism, which is contradict with our goal of speeding up the computation by running on multiple processing units. Furthermore, from hardware development perspective, designing a radix-*R* FNT module with higher value of *R* costs extra resources and increases the computation latency.

From Figure 4.18, radix-4 FNT is able to show a significant improvement (reduced level of decomposition, which directly relates to the reduction of computation time) over radix-2 FNT, but this improvement is diminishing for the case of radix-8 and radix-16 FNTs. Radix-4 FNT shows an optimum point of sacrificing a certain degree of parallelism for lesser FNT decomposition level. Hence, we focus on the implementation using radix-2 and radix-4 FNTs in our work.

#### 4.4.2.1 Radix-2 and radix-4 FNT module

Figure 4.19, 4.19 and 4.20 show the block diagram of our design for a typical radix-2 FNT module, a radix-4 FNT module built by connecting four radix-2 FNT modules in 2-by-2 manner (denotes as 4r2 FNT module for simplicity) and a typical radix-4 FNT module respectively. A radix-2 FNT costs 4 clock cycles to complete. Comparing both of the radix-4 FNT modules as shown in Figure 4.20 and Figure 4.21, the 4r2 FNT module has a latency of 7 clock cycles while the typical radix-4 FNT module costs only 5 clock cycles, provided all of them are running at same clock frequency of 20MHz. In term of hardware resources, the typical radix-4 FNT module also costs one less modular multiplier submodule and two stages lesser pipeline registers.

In short, the typical radix-4 FNT module is not only has lower latency but also required lesser resources. Hence, we will be using this typical radix-4 FNT module for our next implementation.

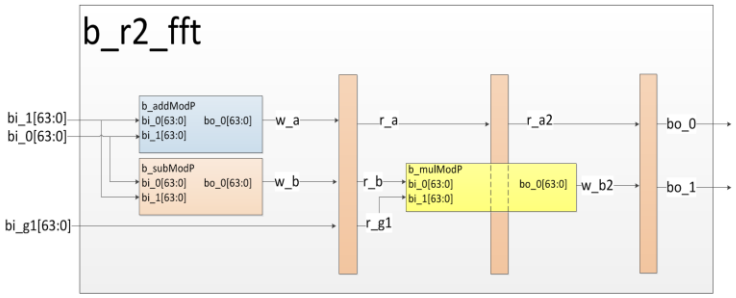


**Figure 4.19 Block diagram: radix-2 FNT module**

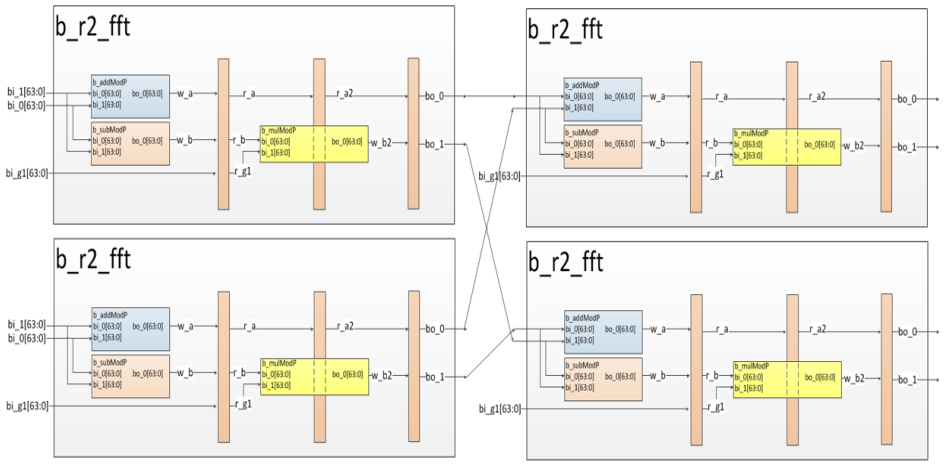


**Figure 4.20 Block diagram: radix-4 FNT module built with (2x2) radix-2 modules**

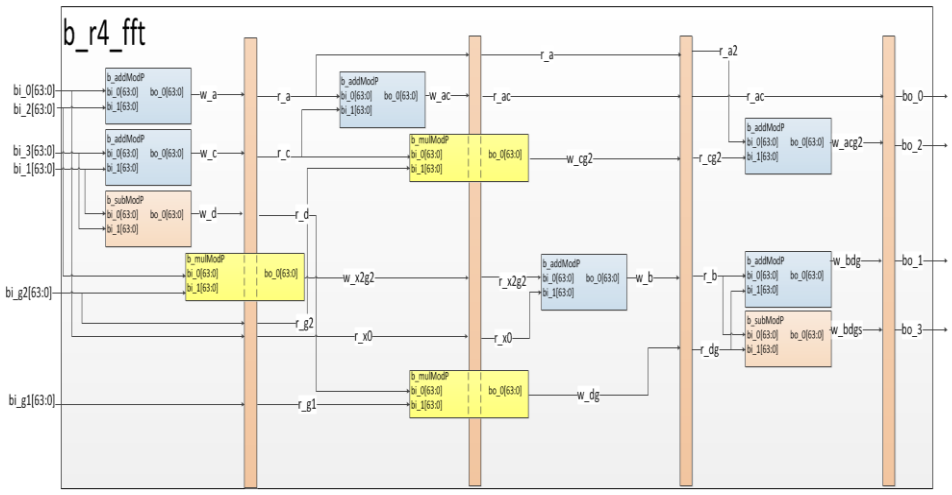


**Figure 4.21 Block diagram: radix-4 FNT module**

#### 4.4.2.2 CTFNT Decomposition

Two levels of CTFNT are used in implementing the 256-point NTT. Both levels of the decomposition are done symmetrically ($N_1 = N_2$) to save the memory required to store the precomputed twiddle factors by sharing them for both the column and row NTTs of CTFNT. The first level of decomposition divides the 256-point NTT into (16 * 16)-point CTFNT, and each of the 16-point NTTs is then divided into (4 * 4)-point CTFNT at the second level decomposition. All of the 4-point NTTs at the second level will be computed using the typical radix-4 CTFNT module discussed in previous section (section 4.4.2.1). Figure 4.22 illustrates the 256-point NTT decomposition used in this implementation.

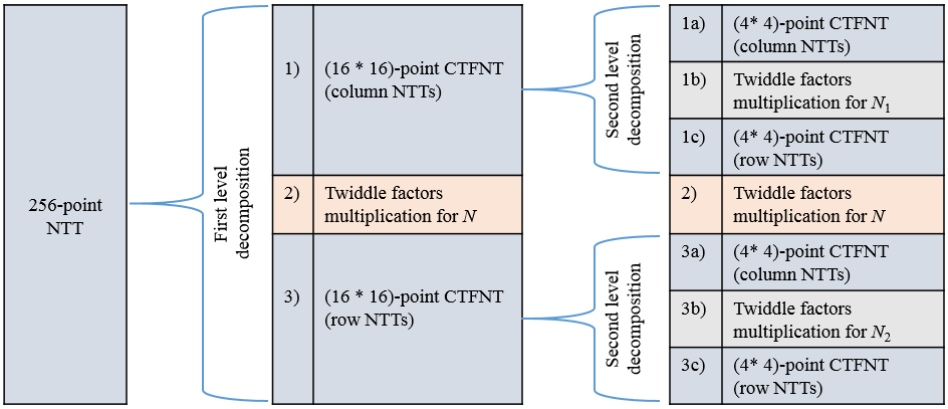


**Figure 4.22 256-point NTT CTFNT decomposition**

Figure 4.23 shows the block diagram of implementing a SSMA module using typical radix-4 FNT module. The functions of each part of the blocks are described as follows:

a) Block RAM, memory unit of the module;
b) Typical radix-4 FNT module, four input and four output;
c) Twiddle factors multipliers;
d) Temporary registers to store the intermediate data of FNT and convolution unit;
e) Typical radix-4 FNT module, four input and four output;
f) Twiddle factors multipliers;

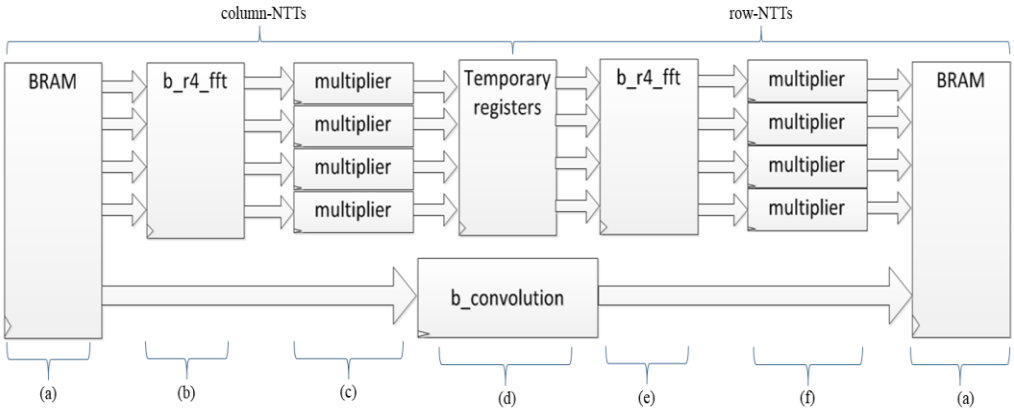


**Figure 4.23 Block diagram: SSMA module**

Part (a), (b) and (c) are used to compute column-NTTs, four points at a time. The results are stored into part (d), the temporary registers and wait until all the column-NTTs before proceed to row-NTTs. The row-NTTs are handled by part (e), (f) and (a). The column-NTTs read data from the block RAM and write to temporary registers; while the row-NTTs read data from the temporary registers and write back to the block RAM. Part (d) compute the convolution of the SSMA by reading and writing the data to and from the block RAM without the needs of intermediate memory.

**Table 4.7 SSMA operation and clock cycles count**

| Steps | Operation | Clock cycles (without pipeline) | Clock cycles (partial pipelined) |
|---|---|---|---|
| 1. | Fetch four NTT data from BRAM. | 1 | 12 |
| 2. | Compute one 4-point column NTT. | 5 | |
| 3. | Multiply the NTT output with twiddle factors. | 2 | |
| 4. | Store the column NTT data into temporary registers. | 1 | |
| 5. | Repeat step (1) to (4) 3 times to complete (4*4)-point column NTT. | 27 | |
| 6. | Fetch four NTT data from temporary registers. | 1 | 12 |
| 7. | Compute one 4-point row NTT. | 5 | |
| 8. | Multiply the NTT output with twiddle factors. | 2 | |
| 9. | Store the row NTT data back to the BRAM. | 1 | |
| 10. | Repeat step (6) to (9) 3 times to complete (4*4)-point row NTT. | 27 | |
| 11. | Repeat step (1) to (10) 15 times to complete (16*16)-point column NTT. | 1080 | 360 |
| 12. | Repeat step (1) to (11) for (16*16)-point row NTT. | 1152 | 384 |
| 13. | Repeat step (1) to (12) for second multiplication operand. | 2304 | 768 |
| 14. | Fetch four NTT data from BRAM. | 1 | 67 |
| 15. | Run convolution of the SSMA, four points at a time. | 2 | |
| 16. | Store results of step (15) back to block RAM. | 1 | |
| 17. | Repeat step (14) to (16) sixty three times to complete full convolution. | 126 | |
| 18. | Repeat step (1) to (12) for inverse NTT. | 2304 | 768 |
| | Total: | 7042 | 2371 |
| | Latency(20MHz): | 352.1ms | 118.55ms |

Table 4.7 show the flow of the multiplication and number of clock cycles needed to complete each of the steps. Table 4.7 states that a total number of 7042 clock cycles are needed to complete SSMA with 256-point NTT when the design is not pipelined. Figure 4.24 shows the comparison of the timing diagram of the pipelined and not pipelined design.

However, the pipelined design discusses in this section is actually “partial-pipelined”. This mean some of the steps can be run concurrently, the clock cycles are overlapped to improve the latency. The column-NTTs, row-NTTs and convolution parts are three different parts of the circuit that are pipelined. The pipelined result is also shown in Table 4.7, where 2371 clock cycles or 118.55ms is needed to complete the SSMA with 256-points. This timing diagram for this partial-pipelined design is shown in Figure 4.25.

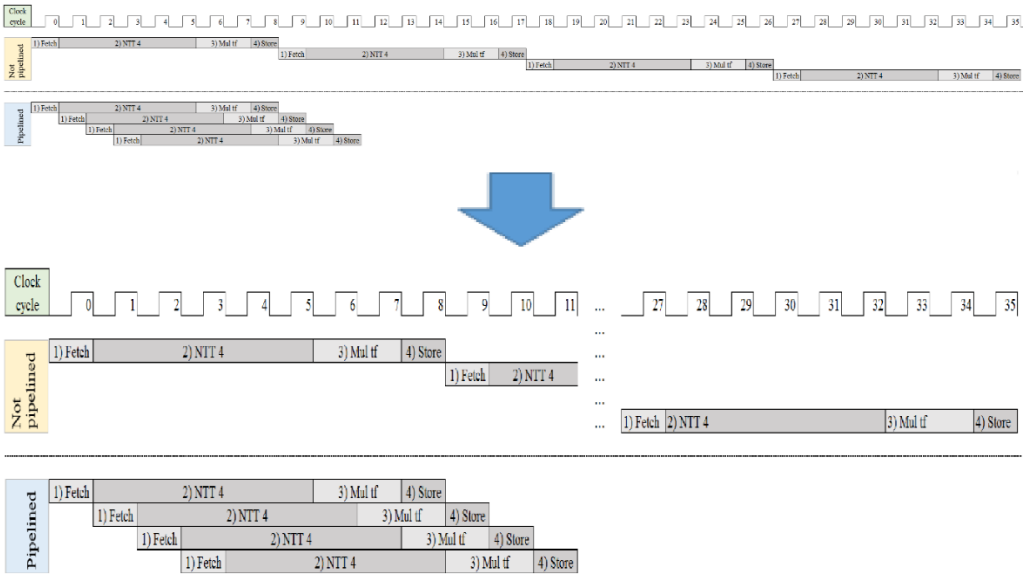


**Figure 4.24 Pipelined and non-pipelined timing comparison**

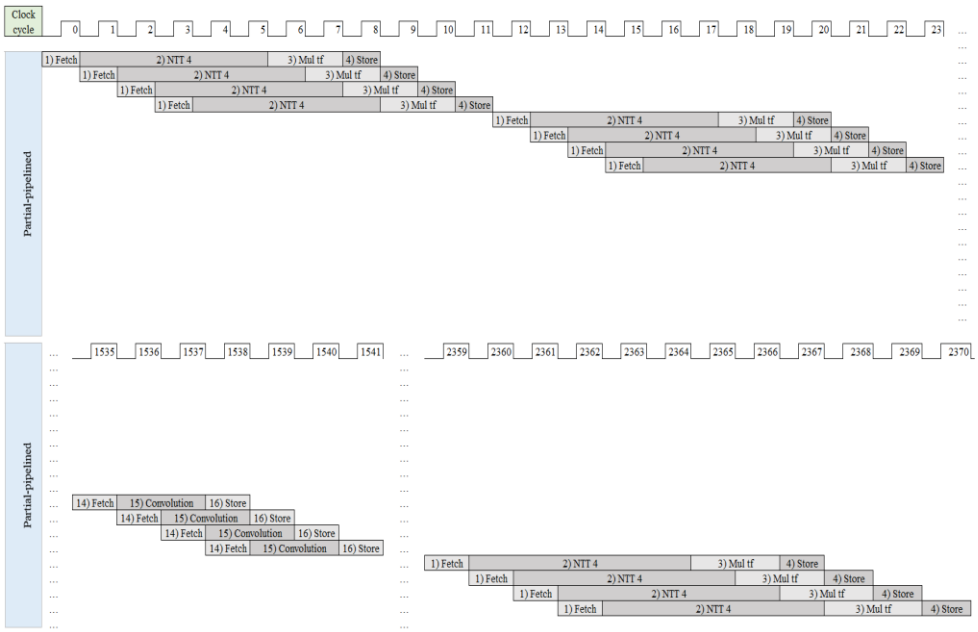


**Figure 4.25 Timing diagram for partial –pipelined design**

#### 4.4.2.2 Findings

The problem of this design is the dependency between the column-NTTs, row-NTTs and convolution, where the row-NTTs have to wait for the column-NTTs to complete before proceed; and the convolution have to wait for both the (16*16)-column-NTTs and (16*16)-row-NTTs to complete before it started. This causes the other two parts have to be idle when one is processing, drastically reduce the hardware occupancy and efficiency. In our next design, we aimed to improve this by modifying the circuit to become fully pipelined.

### 4.4.3 Third Design: SSMA with dedicated Radix-4 CTFNT

#### 4.4.3.1 Improved radix-4 CTFNT

The typical radix-4 FNT module as shown in Figure 4.21 utilizes two 64-bit modular multipliers to perform the twiddle factors multiplication of the 4-points NTT. The twiddle factors used for this radix-4 NTT are constant values of 0x1000000000000, $0x1000000000000^2$ for forward transform and $0x1000000000000^2$, $0x1000000000000^3$ for inverse transform. The multiplication with these twiddle factors can be calculated through left shifting operation for 48-bit, 96-bit and 144-bit respectively, to reduce the hardware resources needed. The details of design is shown in section 4.4.4.7.

#### 4.4.3.2 Timing performance of 16-point NTT

In this work, three different hardware modules are designed to compute 16-point NTT as a preliminary study to the performance.

1. First design: A single radix-2 module is used repeatedly to compute the 16-point NTT;
2. Second design: Four radix-2 modules are instantiated to build a radix-4 module as shown in Figure 3.4 to speed up the computation by reducing memory access to the BRAM at the costs of extra hardware resources;
3. Third design: we propose a novel radix-4 NTT design that is dedicated for the used with CTFNT implementation as discussed in section 4.4.3.1. We denote this module as radix-4 CTFNT module. The drawback of this radix-4 CTFNT module is that the scalability of using it for non-power-of-4 NTT size is sacrificed

**Table 4.8 Resource utilization and timing performance for 16-point NTT**

| Hardware Design | Resources | | | | | Timing | | |
|---|---|---|---|---|---|---|---|---|
| | Look-up Table (LUTs) | Flip-Flop (FF) | LUTRAM | BRAM | DSP | Clock Cycle | Period (ns) | Latency (ns) |
| Radix-2 NTT | 1687 | 456 | 3 | 4 | 12 | 36 | 50 | 1800 |
| Radix-4 NTT | 6662 | 2063 | 68 | 7 | 48 | 20 | 50 | 1000 |
| Radix-4 CTFNT | 6421 | 1805 | 35 | 8 | 45 | 18 | 50 | 900 |

Table 4.8 shows the hardware resources utilization and timing result of three of the designs when used to compute 16-point NTT.

Table 4.8 also indicates that our proposed design is able to outperform a typical radix-4 module (built with four radix-2 modules connected in two-by-two manner as shown in Figure 3.4) in term of both hardware resources utilization and timing performance. Although the basic radix-2 NTT module costs lesser resources compared to our proposed design, our proposed design is able to compute 16-point NTT with only half of the computation time. This is a reasonable tradeoff between hardware resources and timing performance.

#### 4.4.3.3 Multiple radix-4 column/row-NTTs design

To remove the dependency between the column and row NTTs and convolution in the design as discussed in section 4.4.2, we modify the design from a sequential column-NTTs to row-NTTs to a parallel multiple radix-4 column/row-NTTs design, where the set of typical radix-4 FNT modules can be configured to perform either column or row NTTs. The design is shown in Figure 4.26.

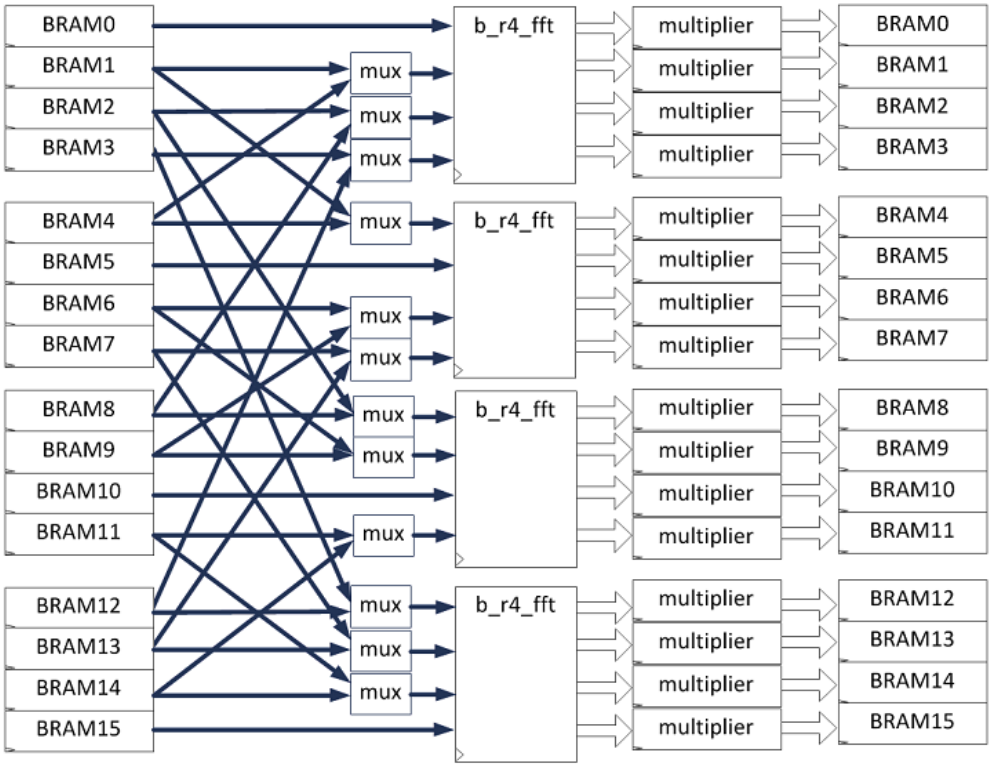


**Figure 4.26 Block diagram: multiple radix-4 column/row-NTTs design**

Four typical radix-4 FNT modules are instantiated and arranged in parallel as shown in Figure 4.26 to allow four 4-point NTTs to be computed at the same time. Multiplexers are used to select the input either from the local set of BRAM or the other sets. The computed 4-point NTTs results are passed into multipliers for twiddle factors multiplication before storing back to the BRAM.

However, to allow four of the radix-4 FNT modules to compute column-NTTs or row-NTTs, extra control signals are needed to route the data before storing them back to the BRAM. Figure 4.27 shows the block diagram of the design with extra control added.

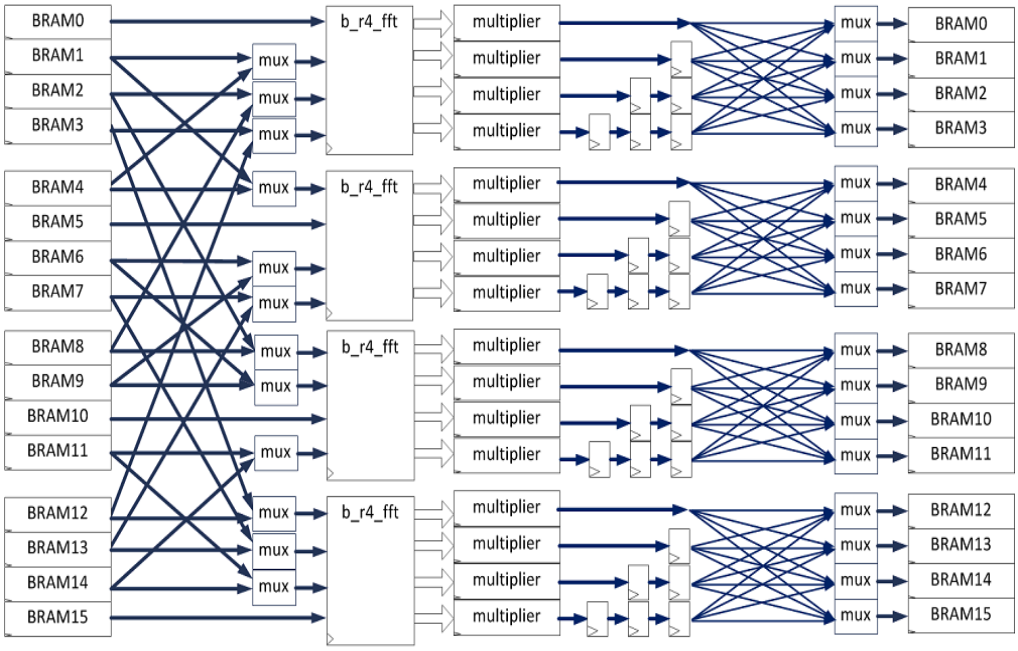


**Figure 4.27 Block diagram: multiple radix-4 column/row-NTTs design (modified)**

Figure 4.27 shows the block diagram from Figure 4.27 modified with extra control signals. Extra pipeline registers are also added to the output of the multipliers to avoid collision of data when two or more data are trying to write into the same BRAM. For instance, when four of the data from the multipliers are trying to write into BRAM0, the second data will be delayed by one clock cycle, the third one is delayed by two clock cycles and three clock cycles for the fourth one. Figure 4.28 illustrates the sequences of the data flow, where the input are fed in to the module in order and the output are delayed. The delays costs

three clock cycles to fill up the pipeline fully but this tradeoff is insignificant as the full SSMA took 198 clock cycles to compute.

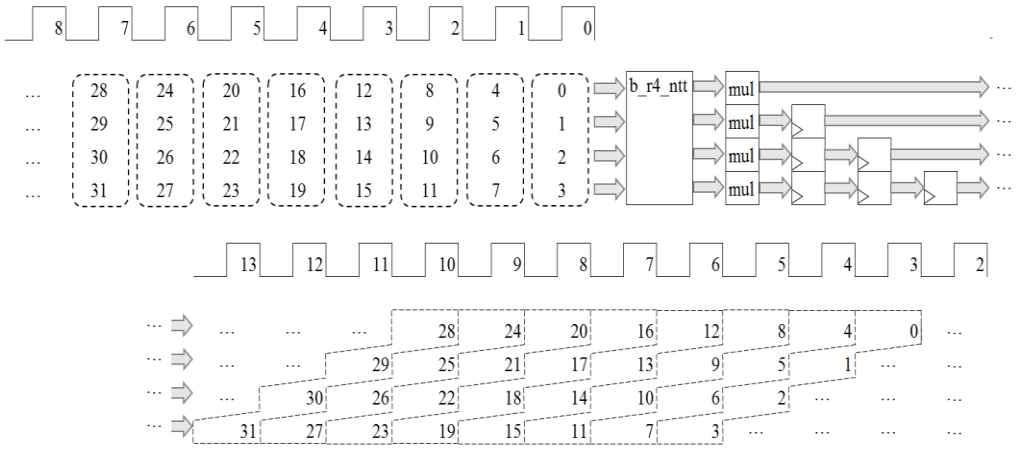


**Figure 4.28 Timing diagram for data flow control**

The details of this radix-4 CTFNT module is shown in section 4.4.4.7, denotes as b_r4_ntt. Four of the NTT output from this module are connected to one multiplier each to form b_r4_ntt_m in section 4.4.4.8. These multipliers are used for twiddle factors multiplication (step 1b), 2) and 3b) shown in Figure 4.29) and the convolution in SSMA. Next, four of these b_r4_ntt_m modules are instantiated as sub-modules to create b_4r4_ntt_m (section 4.4.4.9). If we describe the process between reading data from BRAM and writing back to it as a loop, this b_4r4_ntt_m is able to compute one level of (4 * 4)-point CTFNT multiplication during each loop. Figure 4.29 illustrates the flow of how a full 3072-bit SSMA can be perform using this module as an extension from Figure 4.22.

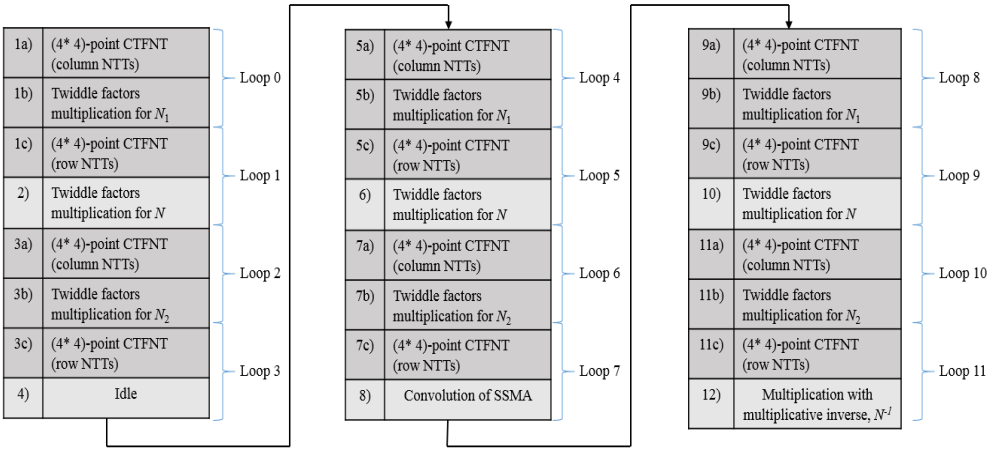


**Figure 4.29 SSMA design flow**

Every two subsequent steps (including sub-steps) from Figure 4.29 are computed in one loop, where the first step is the 4-point NTT followed by multiplication. This multiplication can be twiddle factors multiplication, convolution or multiplicative inverse multiplication. Step 1a) to 3c) are the process to compute forward NTT of first multiplication operand, the forward NTT of second multiplication operand is computed from step 5a) to 7c). Step 9a) to 12) compute the inverse NTT for the multiplication product. A total number of 12 loops are needed to complete a 3072-bit SSMA. Note that step 4) is an idle step. This idle operation is done by performing multiplication with one to the respective data. Extra control logic can be implemented into the design to bypass this step but only two clock cycles can be save from the multiplication which is not worth of the hardware resources. The experimental results of this design is shown in section 5.3.

### 4.4.4 Hardware Module

This section provides the details of all the hardware modules in this implementation with interface diagrams and table to describe the functionalities including each of the input output pins of each of the modules.

#### 4.4.4.1 Design Overview

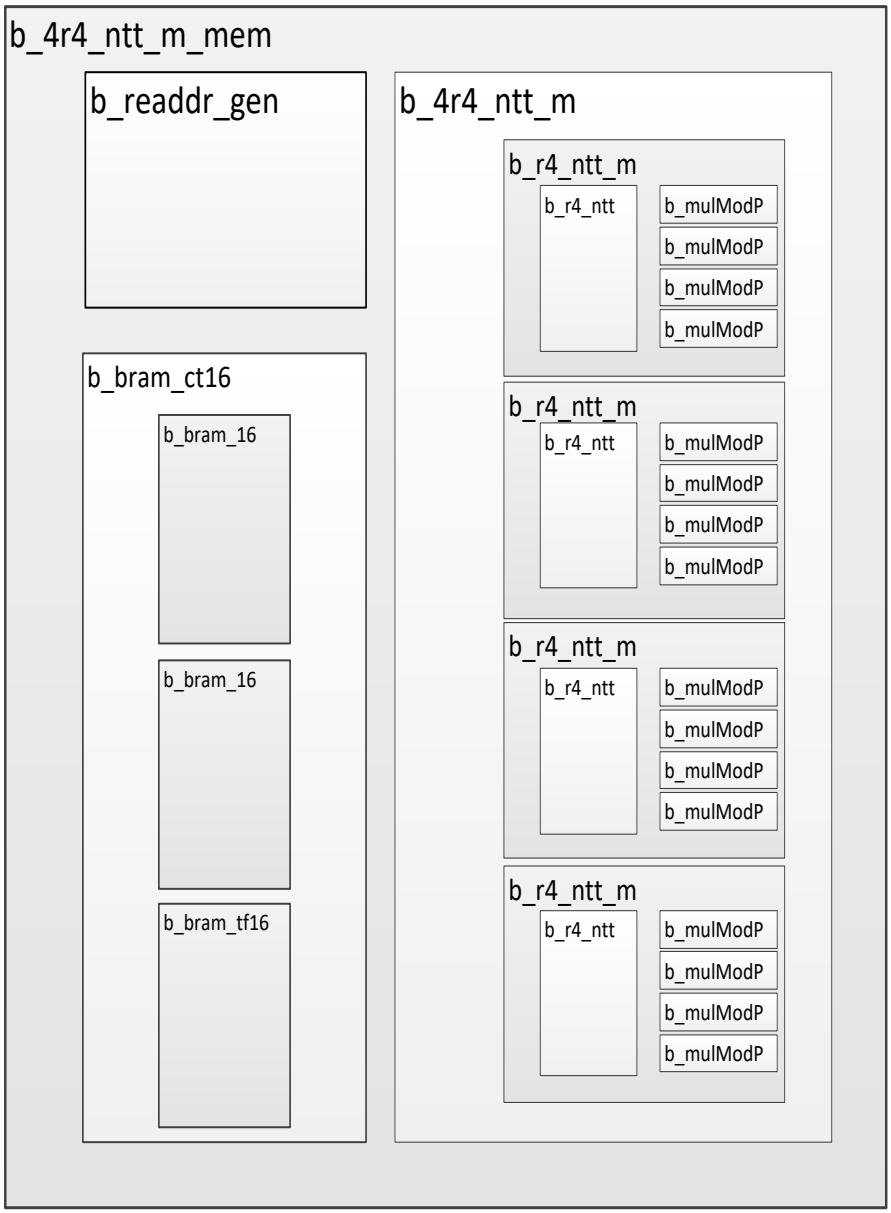


**Figure 4.30 Design overview**

**Modular arithmetic units (b_addModP, b_subModP, b_shl48ModP and b_shl96ModP) are hidden from Figure 4.24 for readability.*

#### 4.4.4.2 b_addModP

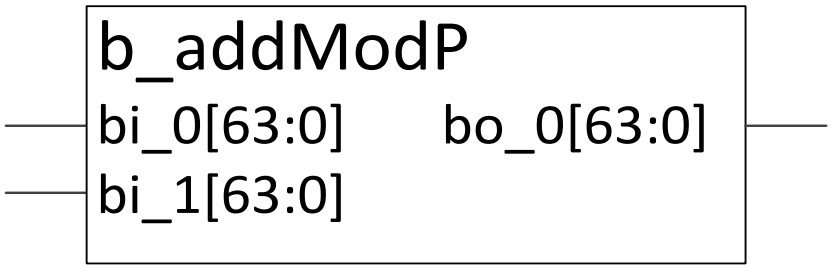


**Figure 4.29 Interface diagram: 64-bit modular adder**

**Table 4.9 Description: 64-bit modular adder**

| Module name | | | |
|---|---|---|---|
| b_addModP | | | |
| Functionality | | | |
| Perform modular addition of two 64-bit operands, with modulus = 0xFFFFFFFF00000001. | | | |
| Pin name | Direction | Width (Bits) | Function |
| bo_0 | Output | 64 | Sum of modular addition. |
| bi_0 | Input | 64 | First operand of modular addition. |
| bi_1 | Input | 64 | Second operand of modular addition. |

#### 4.4.4.3 b_subModP

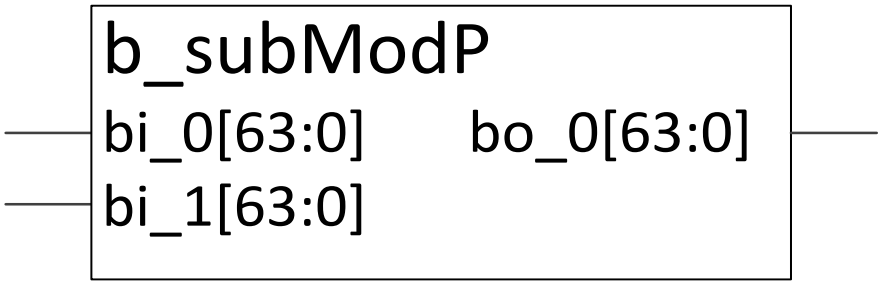


**Figure 4.30 Interface diagram: 64-bit modular subtractor**

**Table 4.10 Description: 64-bit modular subtractor**

| Module name | | | |
|---|---|---|---|
| b_subModP | | | |
| Functionality | | | |
| Perform modular subtraction of two 64-bit operands, with modulus = 0xFFFFFFFF00000001. | | | |
| Pin name | Direction | Width (Bits) | Function |
| bo_0 | Output | 64 | Difference of modular subtraction. |
| bi_0 | Input | 64 | First operand of modular subtraction. |
| bi_1 | Input | 64 | Second operand of modular subtraction. |

#### 4.4.4.4 b_shl48ModP

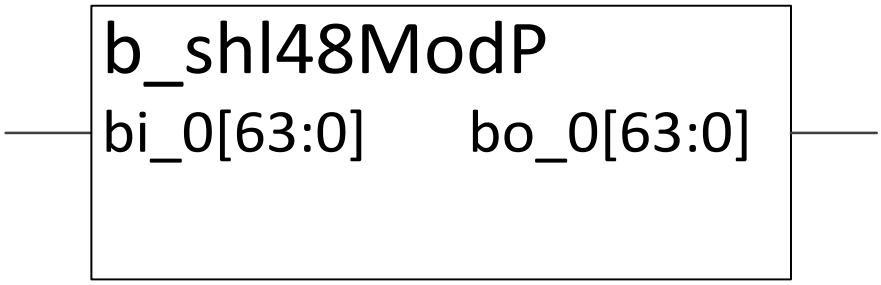


**Figure 4.31 Interface diagram: 64-bit modular 48-bit left shifter**

**Table 4.11 Description: 64-bit modular 48-bit left shifter**

| Module name | | | |
|---|---|---|---|
| b_shl48ModP | | | |
| Functionality | | | |
| Perform modular left shift 48-bit of input 64-bit operand, with modulo = 0xFFFFFFFF00000001. Dedicated for the use of twiddle factor multiplication for 4-points NTT for twiddle factor = 0x1000000000000. | | | |
| Pin name | Direction | Width (Bits) | Function |
| bo_0 | Output | 64 | Result of modular left shift. |
| bi_0 | Input | 64 | Input operand of modular left shift. |

#### 4.4.4.5 b_shl96ModP

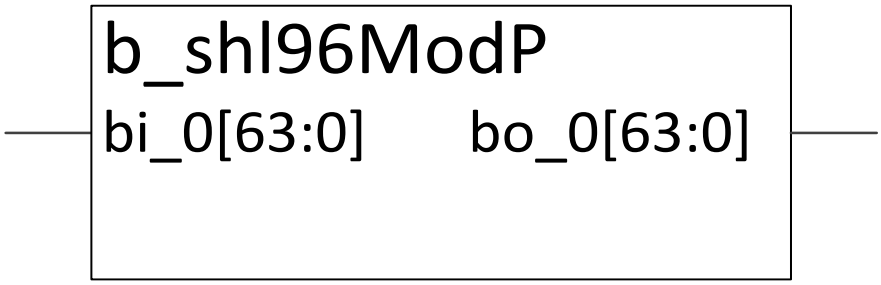


**Figure 4.32 Interface diagram: 64-bit modular 96-bit left shifter**

**Table 4.12 Description: 64-bit modular 96-bit left shifter**

| Module name | | | |
|---|---|---|---|
| b_shl96ModP | | | |
| Functionality | | | |
| Perform modular left shift 96-bit of input 64-bit operand, with modulo = 0xFFFFFFFF00000001. Dedicated for the use of twiddle factor multiplication for 4-points NTT for twiddle factor = $(0x1000000000000)^2$. | | | |
| Pin name | Direction | Width (Bits) | Function |
| bo_0 | Output | 64 | Result of modular left shift. |
| bi_0 | Input | 64 | Input operand of modular left shift. |

#### 4.4.4.6 b_mulModP

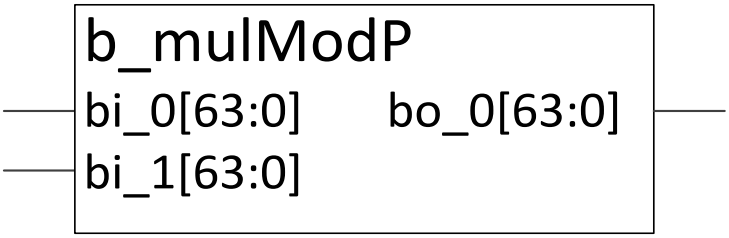


**Figure 4.33 Interface diagram: 64-bit modular multiplier**

**Table 4.13 Description: 64-bit modular multipler**

| Module name | | | |
|---|---|---|---|
| b_mulModP | | | |
| Functionality | | | |
| Perform modular multiplication of two 64-bit operands, with modulus = 0xFFFFFFFF00000001 in two clock cycles. | | | |
| Pin name | Direction | Width (Bits) | Function |
| bo_0 | Output | 64 | Product of modular multiplication. |
| bi_0 | Input | 64 | First operand of modular multiplication. |
| bi_1 | Input | 64 | Second operand of modular multiplication. |

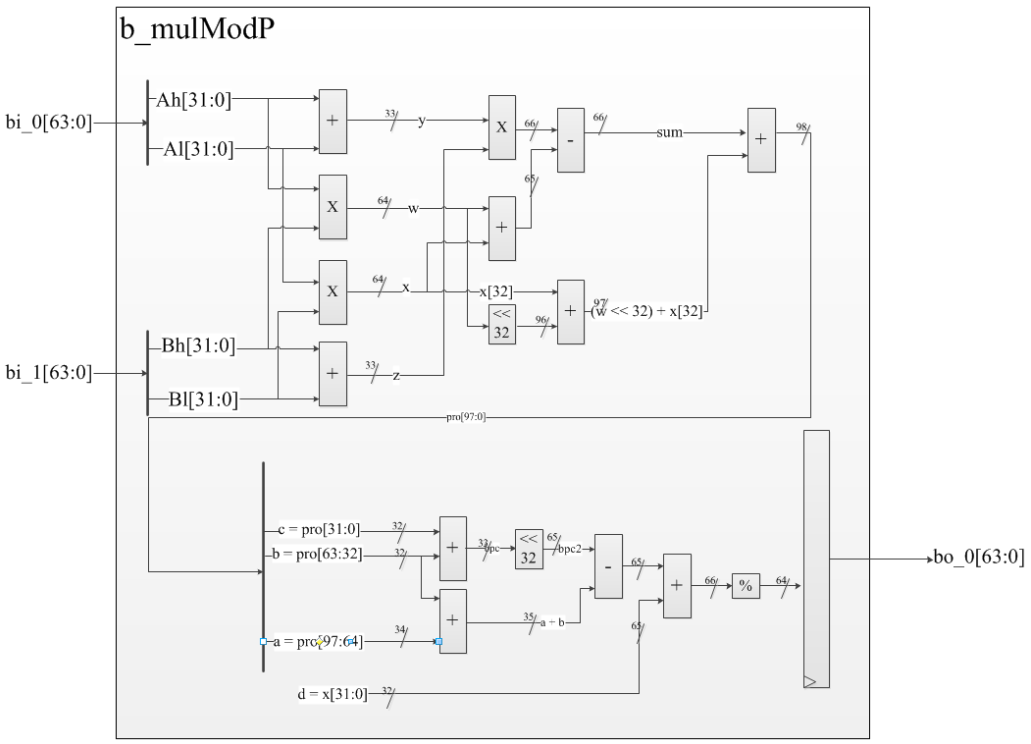


**Figure 4.34 Internal Interface diagram: 64-bit modular multiplier**

#### 4.4.4.7 b_r4_ntt

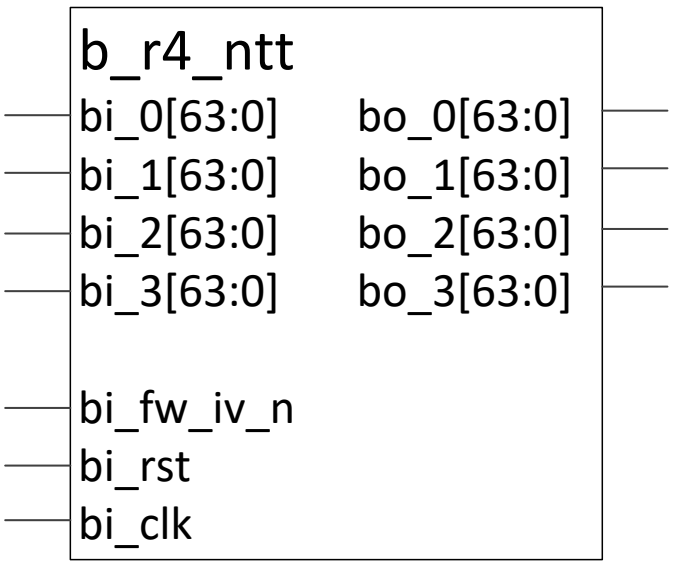


**Figure 4.35 Interface diagram: 4-point NTT processing unit**

**Table 4.14 Description: 4-point NTT processing unit**

| Module name | | | |
|---|---|---|---|
| b_r4_ntt | | | |
| Functionality | | | |
| Compute 4-points NTT, perform forward transform or inverse transform based on selected mode. The output with larger indices are designed to be one clock cycle later than smaller output to meet the requirements for the memory access patterns. | | | |
| Pin name | Direction | Width (Bits) | Function |
| bo_0 | Output | 64 | First output of transformation. |
| bo_1 | Output | 64 | Second output of transformation. |
| bo_2 | Output | 64 | Third output of transformation. |
| bo_3 | Output | 64 | Forth output of transformation. |
| bi_0 | Input | 64 | First input of transformation. |
| bi_1 | Input | 64 | Second input of transformation. |
| bi_2 | Input | 64 | Third input of transformation. |
| bi_3 | Input | 64 | Forth input of transformation. |

**Continued from Table 4.14**

| Pin name | Direction | Width (Bits) | Function |
|---|---|---|---|
| bi_fw_iv_n | Input | 1 | Select transformation mode:<br>0: Inverse transform.<br>1: Forward transform. |
| bi_clk | Input | 1 | Global clock signal. |
| bi_rst | Input | 1 | Global reset signal. |

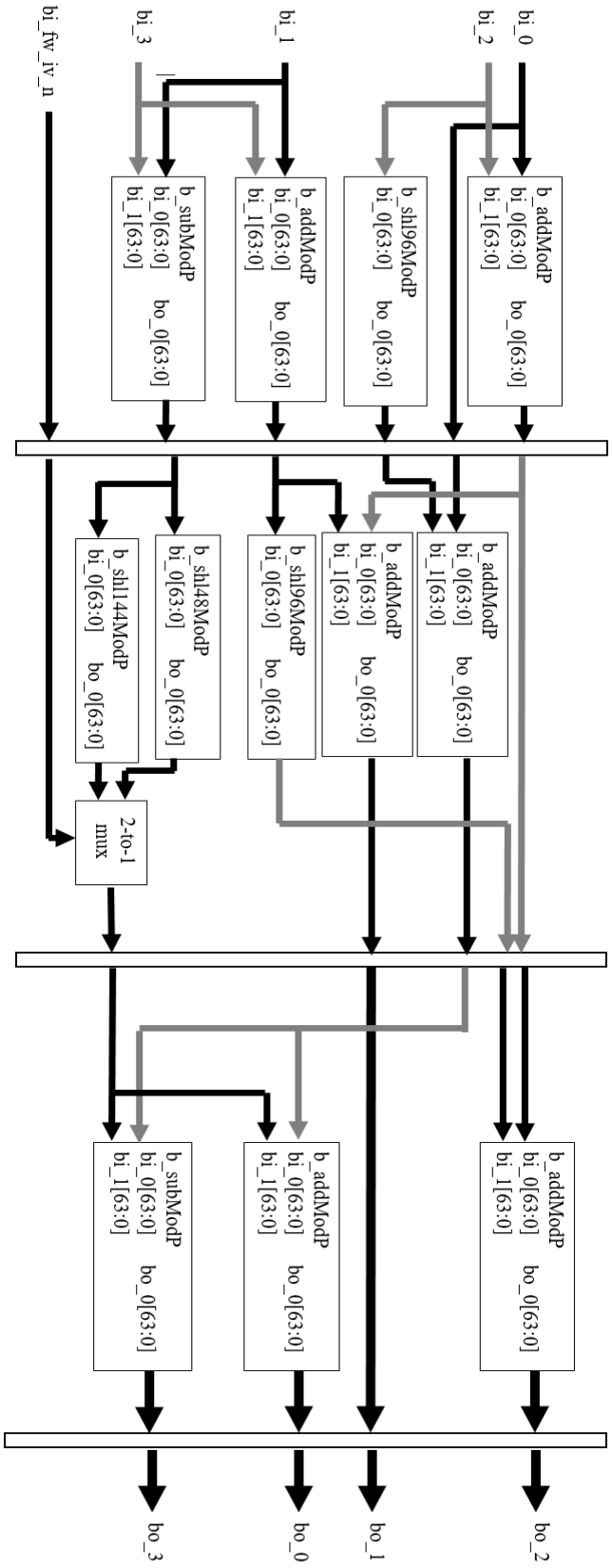


**Figure 4.36 Internal Interface diagram: 4-point NTT processing unit**

#### 4.4.4.8 b_r4_ntt_m

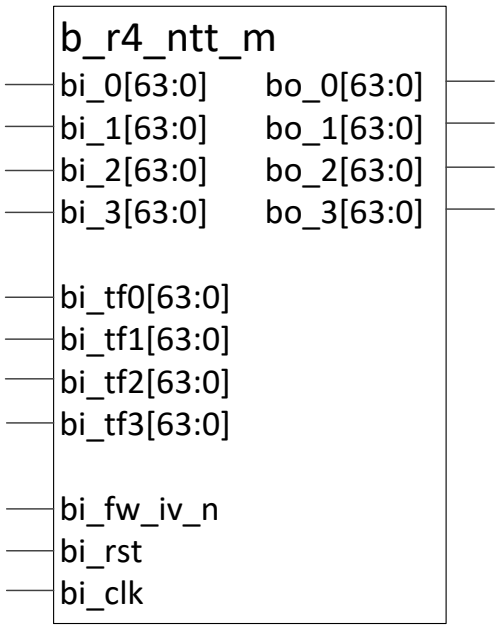


**Figure 4.37 Interface diagram: 4-point NTT processing unit with multiplier**

**Table 4.15 Description: 4-point NTT processing unit with multiplier**

| Module name | | | |
|---|---|---|---|
| b_r4_ntt_m | | | |
| Functionality | | | |
| Compute 4-points NTT with twiddle factors multiplication. Built by connecting four b_mulModP to each of the output of b_r4_ntt. | | | |
| Pin name | Direction | Width (Bits) | Function |
| bo_0 | Output | 64 | First output of transformation. |
| bo_1 | Output | 64 | Second output of transformation. |
| bo_2 | Output | 64 | Third output of transformation. |
| bo_3 | Output | 64 | Forth output of transformation. |
| bi_0 | Input | 64 | First input of transformation. |
| bi_1 | Input | 64 | Second input of transformation. |
| bi_2 | Input | 64 | Third input of transformation. |

**Continued from Table 4.15**

| Pin name | Direction | Width (Bits) | Function |
|---|---|---|---|
| bi_3 | Input | 64 | Forth input of transformation. |
| bi_tf0 | Input | 64 | First twiddle factor input. |
| bi_tf1 | Input | 64 | Second twiddle factor input. |
| bi_tf2 | Input | 64 | Third twiddle factor input. |
| bi_tf3 | Input | 64 | Forth twiddle factor input. |
| bi_fw_iv_n | Input | 1 | Select transformation mode:<br>0: Inverse transform.<br>1: Forward transform. |
| bi_clk | Input | 1 | Global clock signal. |
| bi_rst | Input | 1 | Global reset signal. |

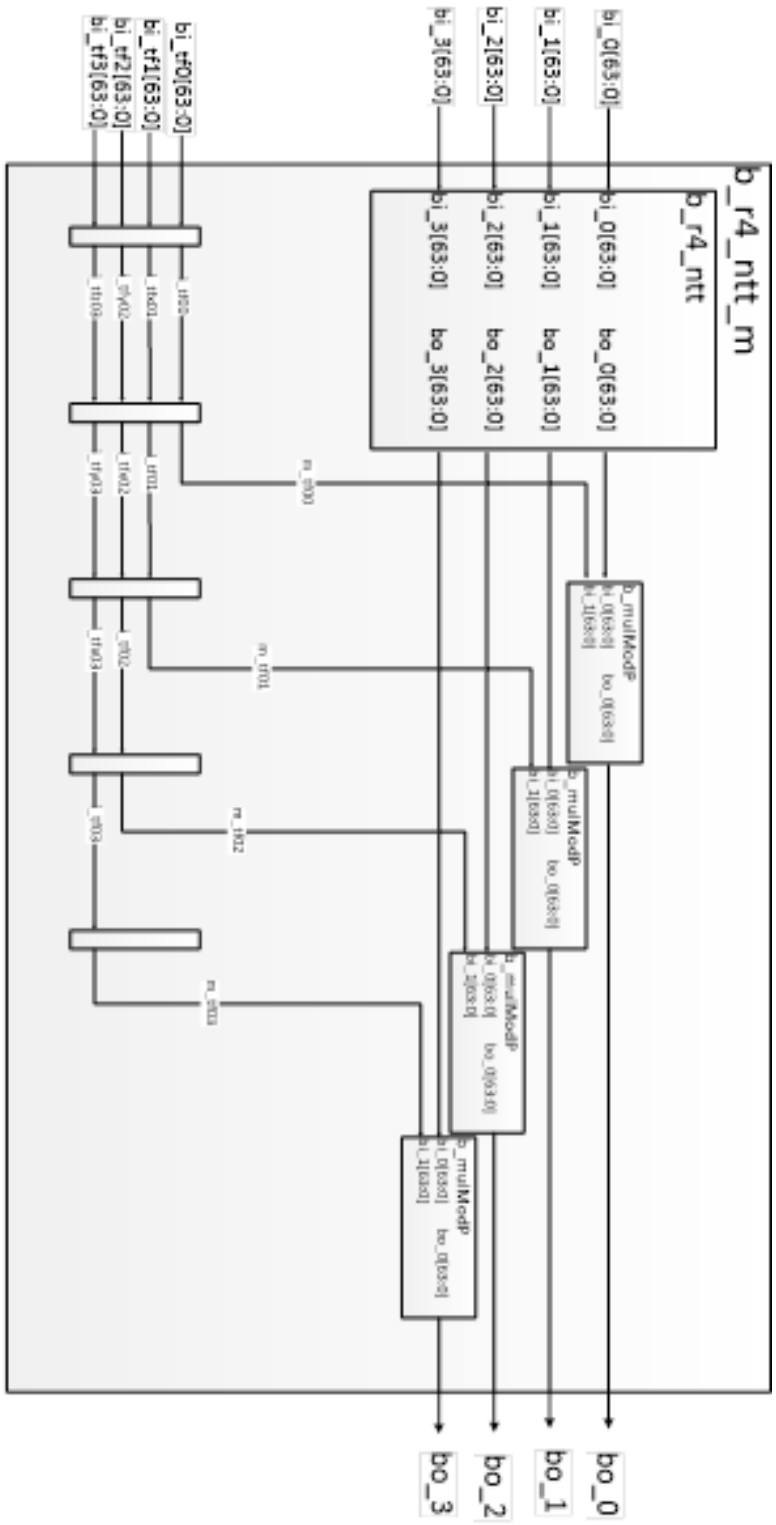


**Figure 4.37 Internal Interface diagram: 4-point NTT processing unit with multiplier**

#### 4.4.4.9 b_4r4_ntt_m

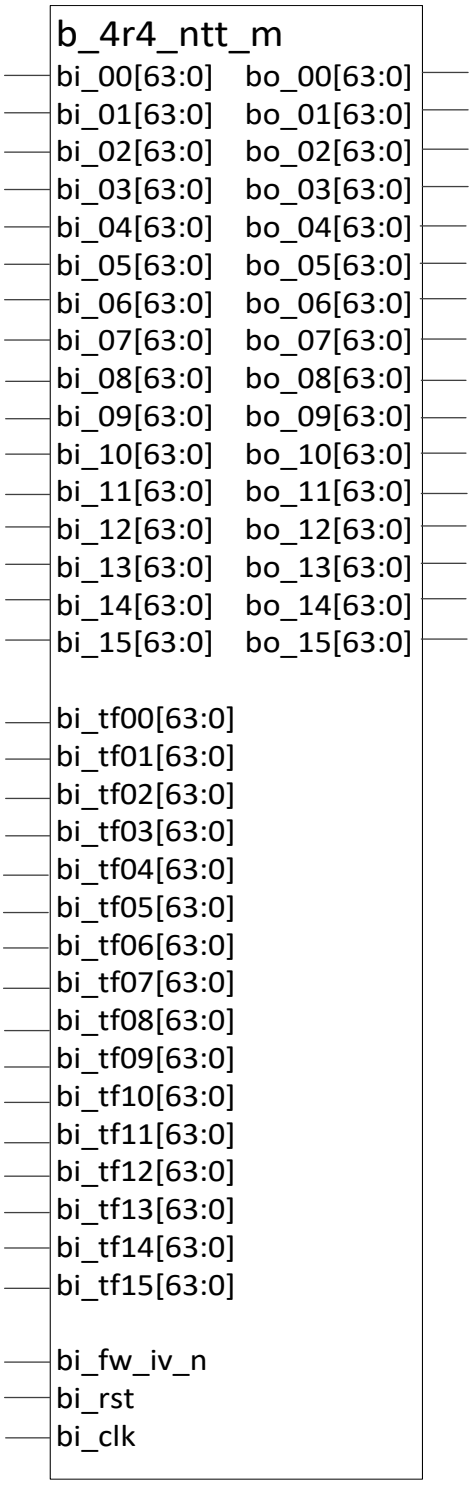


**Figure 4.38 Interface diagram: 16-point NTT processing unit**

**Table 4.16 Description: 16-point NTT processing unit**

| Module name | | | |
|---|---|---|---|
| b_4r4_ntt_m | | | |
| Functionality | | | |
| Compute four 4-points NTTs with twiddle factors multiplication. Compose of four units of b_r4_ntt. | | | |
| Pin name | Direction | Width (Bits) | Function |
| bo_00 | Output | 64 | First output of first 4-points NTT. |
| bo_01 | Output | 64 | Second output of first 4-points NTT. |
| bo_02 | Output | 64 | Third output of first 4-points NTT. |
| bo_03 | Output | 64 | Forth output of first 4-points NTT. |
| bo_04 | Output | 64 | First output of second 4-points NTT. |
| bo_05 | Output | 64 | Second output of second 4-points NTT. |
| bo_06 | Output | 64 | Third output of second 4-points NTT. |
| bo_07 | Output | 64 | Forth output of second 4-points NTT. |
| bo_08 | Output | 64 | First output of third 4-points NTT. |
| bo_09 | Output | 64 | Second output of third 4-points NTT. |
| bo_10 | Output | 64 | Third output of third 4-points NTT. |
| bo_11 | Output | 64 | Forth output of third 4-points NTT. |
| bo_12 | Output | 64 | First output of forth 4-points NTT. |
| bo_13 | Output | 64 | Second output of forth 4-points NTT. |
| bo_14 | Output | 64 | Third output of forth 4-points NTT. |
| bo_15 | Output | 64 | Forth output of forth 4-points NTT. |
| bi_00 | Input | 64 | First input of first 4-points NTT. |
| bi_01 | Input | 64 | Second input of first 4-points NTT. |

**Continued from Table 4.16**

| Pin name | Direction | Width (Bits) | Function |
|---|---|---|---|
| bi_02 | Input | 64 | Third input of first 4-points NTT. |
| bi_03 | Input | 64 | Forth input of first 4-points NTT. |
| bi_04 | Input | 64 | First input of second 4-points NTT. |
| bi_05 | Input | 64 | Second input of second 4-points NTT. |
| bi_06 | Input | 64 | Third input of second 4-points NTT. |
| bi_07 | Input | 64 | Forth input of second 4-points NTT. |
| bi_08 | Input | 64 | First input of third 4-points NTT. |
| bi_09 | Input | 64 | Second input of third 4-points NTT. |
| bi_10 | Input | 64 | Third input of third 4-points NTT. |
| bi_11 | Input | 64 | Forth input of third 4-points NTT. |
| bi_12 | Input | 64 | First input of forth 4-points NTT. |
| bi_13 | Input | 64 | Second input of forth 4-points NTT. |
| bi_14 | Input | 64 | Third input of forth 4-points NTT. |
| bi_15 | Input | 64 | Forth input of forth 4-points NTT. |
| bi_tf00 | Input | 64 | First twiddle factor of first 4-points NTT. |
| bi_tf01 | Input | 64 | Second twiddle factor of first 4-points NTT. |
| bi_tf02 | Input | 64 | Third twiddle factor of first 4-points NTT. |
| bi_tf03 | Input | 64 | Forth twiddle factor of first 4-points NTT. |
| bi_tf04 | Input | 64 | First twiddle factor of second 4-points NTT. |
| bi_tf05 | Input | 64 | Second twiddle factor of second 4-points NTT. |
| bi_tf06 | Input | 64 | Third twiddle factor of second 4-points NTT. |
| bi_tf07 | Input | 64 | Forth twiddle factor of second 4-points NTT. |

**Continued from Table 4.16**

| Pin name | Direction | Width (Bits) | Function |
|---|---|---|---|
| bi_tf08 | Input | 64 | First twiddle factor of third 4-points NTT. |
| bi_tf09 | Input | 64 | Second twiddle factor of third 4-points NTT. |
| bi_tf10 | Input | 64 | Third twiddle factor of third 4-points NTT. |
| bi_tf11 | Input | 64 | Forth twiddle factor of third 4-points NTT. |
| bi_tf12 | Input | 64 | First twiddle factor of forth 4-points NTT. |
| bi_tf13 | Input | 64 | Second twiddle factor of forth 4-points NTT. |
| bi_tf14 | Input | 64 | Third twiddle factor of forth 4-points NTT. |
| bi_tf15 | Input | 64 | Forth twiddle factor of forth 4-points NTT. |
| bi_fw_iv_n | Input | 1 | Select transformation mode:<br>0: Inverse transform.<br>1: Forward transform. |
| bi_clk | Input | 1 | Global clock signal. |
| bi_rst | Input | 1 | Global reset signal. |

#### 4.4.4.10 b_4r4_ntt_m_mem

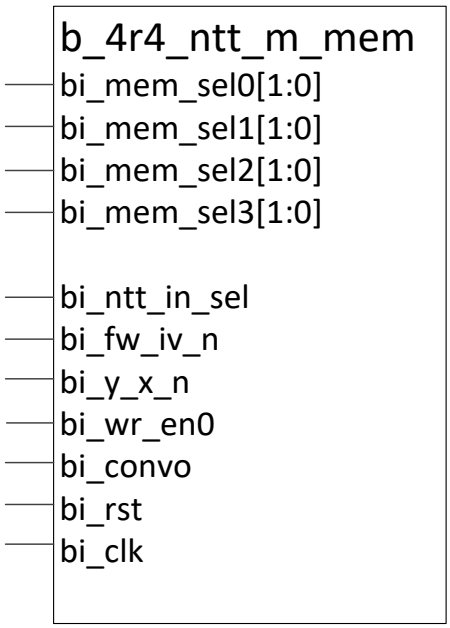


**Figure 4.39 Interface diagram: 16-point NTT processing unit with memory unit**

**Table 4.17 Description: 16-point NTT processing unit with memory unit**

| Module name |
|---|
| b_4r4_ntt_m_mem |
| Functionality |
| Compute four 4-points NTTs with twiddle factors multiplication at once. Reading the data from the memory and writing them back to the memory after execution, repeat until a 256-points NTT or a full SSMA is computed. Compose of the main execution unit, b_4r4_ntt with the main memory unit, b_bram_ct16. |

**Continued from Table 4.17**

| Pin name | Direction | Width (Bits) | Function |
|---|---|---|---|
| bi_mem_in_sel0 | Input | 2 | Select signal for first set memory input:<br>00: Input from NTT module first output.<br>01: Input from NTT module second output.<br>10: Input from NTT module third output.<br>11: Input from NTT module forth output. |
| bi_mem_in_sel1 | Input | 2 | Select signal for second set memory input:<br>00: Input from NTT module first output.<br>01: Input from NTT module second output.<br>10: Input from NTT module third output.<br>11: Input from NTT module forth output. |
| bi_mem_in_sel2 | Input | 2 | Select signal for third set memory input:<br>00: Input from NTT module first output.<br>01: Input from NTT module second output.<br>10: Input from NTT module third output.<br>11: Input from NTT module forth output. |

**Continued from Table 4.17**

| Pin name | Direction | Width (Bits) | Function |
|---|---|---|---|
| bi_mem_in_sel3 | Input | 2 | Select signal for forth set memory input:<br>00: Input from NTT module first output.<br>01: Input from NTT module second output.<br>10: Input from NTT module third output.<br>11: Input from NTT module forth output. |
| bi_ntt_in_sel | Input | 1 | Select signal for NTT input mode:<br>0: Normal.<br>1: Shuffle. |
| bi_fw_iv_n | Input | 1 | Select transformation mode:<br>0: Inverse transform.<br>1: Forward transform. |
| bi_y_x_n | Input | 1 | Select signal for read memory:<br>0: BRAM for operand X.<br>1: BRAM for operand Y. |
| bi_wr_en0 | Input | 1 | Enable signal for write memory:<br>0: Disable.<br>1: Enable. |
| bi_convo | Input | 1 | Select multiplication mode:<br>0: Twiddle factors.<br>1: Convolution. |
| bi_clk | Input | 1 | Global clock signal. |
| bi_rst | Input | 1 | Global reset signal. |

#### 4.4.4.11 b_bram_16

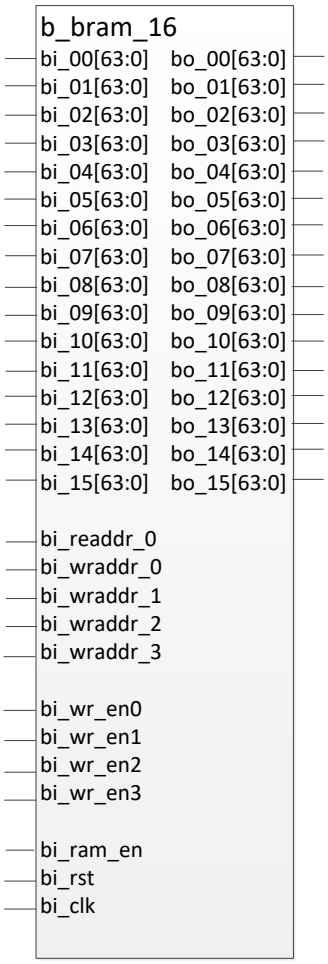


**Figure 4.40 Interface diagram: Memory unit for operands**

**Table 4.18 Description: Memory unit for operands**

| Module name | | | |
|---|---|---|---|
| b_bram_16 | | | |
| Functionality | | | |
| Memory unit to store operands with sixteen input and sixteen output. Implemented with sixteen block RAM, each consists of sixteen registers with 64-bit each. | | | |
| Pin name | Direction | Width (Bits) | Function |
| bo_00 | Output | 64 | First output of first 4-points NTT. |
| bo_01 | Output | 64 | Second output of first 4-points NTT. |
| bo_02 | Output | 64 | Third output of first 4-points NTT. |

**Continued from Table 4.18**

| Pin name | Direction | Width (Bits) | Function |
|---|---|---|---|
| bo_03 | Output | 64 | Forth output of first 4-points NTT. |
| bo_04 | Output | 64 | First output of second 4-points NTT. |
| bo_05 | Output | 64 | Second output of second 4-points NTT. |
| bo_06 | Output | 64 | Third output of second 4-points NTT. |
| bo_07 | Output | 64 | Forth output of second 4-points NTT. |
| bo_08 | Output | 64 | First output of third 4-points NTT. |
| bo_09 | Output | 64 | Second output of third 4-points NTT. |
| bo_10 | Output | 64 | Third output of third 4-points NTT. |
| bo_11 | Output | 64 | Forth output of third 4-points NTT. |
| bo_12 | Output | 64 | First output of forth 4-points NTT. |
| bo_13 | Output | 64 | Second output of forth 4-points NTT. |
| bo_14 | Output | 64 | Third output of forth 4-points NTT. |
| bo_15 | Output | 64 | Forth output of forth 4-points NTT. |
| bi_00 | Input | 64 | First input of first 4-points NTT. |
| bi_01 | Input | 64 | Second input of first 4-points NTT. |
| bi_02 | Input | 64 | Third input of first 4-points NTT. |
| bi_03 | Input | 64 | Forth input of first 4-points NTT. |
| bi_04 | Input | 64 | First input of second 4-points NTT. |
| bi_05 | Input | 64 | Second input of second 4-points NTT. |
| bi_06 | Input | 64 | Third input of second 4-points NTT. |
| bi_07 | Input | 64 | Forth input of second 4-points NTT. |
| bi_08 | Input | 64 | First input of third 4-points NTT. |

**Continued from Table 4.18**

| Pin name | Direction | Width (Bits) | Function |
|---|---|---|---|
| bi_09 | Input | 64 | Second input of third 4-points NTT. |
| bi_10 | Input | 64 | Third input of third 4-points NTT. |
| bi_11 | Input | 64 | Forth input of third 4-points NTT. |
| bi_12 | Input | 64 | First input of forth 4-points NTT. |
| bi_13 | Input | 64 | Second input of forth 4-points NTT. |
| bi_14 | Input | 64 | Third input of forth 4-points NTT. |
| bi_15 | Input | 64 | Forth input of forth 4-points NTT. |
| bi_readdr_0 | Input | 4 | Read address for the memory output. |
| bi_wraddr_0 | Input | 4 | Write address for first set of data. |
| bi_wraddr_1 | Input | 4 | Write address for second set of data. |
| bi_wraddr_2 | Input | 4 | Write address for third set of data. |
| bi_wraddr_3 | Input | 4 | Write address for forth set of data. |
| bi_wr_en0 | Input | 1 | Write enable signal for first set of memory:<br>0: Disable.<br>1: Enable. |
| bi_wr_en1 | Input | 1 | Write enable signal for second set of memory:<br>0: Disable.<br>1: Enable. |
| bi_wr_en2 | Input | 1 | Write enable signal for third set of memory:<br>0: Disable.<br>1: Enable. |
| bi_wr_en3 | Input | 1 | Write enable signal for forth set of memory:<br>0: Disable.<br>1: Enable. |

**Continued from Table 4.18**

| Pin name | Direction | Width (Bits) | Function |
|---|---|---|---|
| bi_ram_en | Input | 1 | Enable signal for the entire unit:<br>0: Disable.<br>1: Enable. |
| bi_clk | Input | 1 | Global clock signal. |
| bi_rst | Input | 1 | Global reset signal. |

#### 4.4.4.12 b_bram_tf16

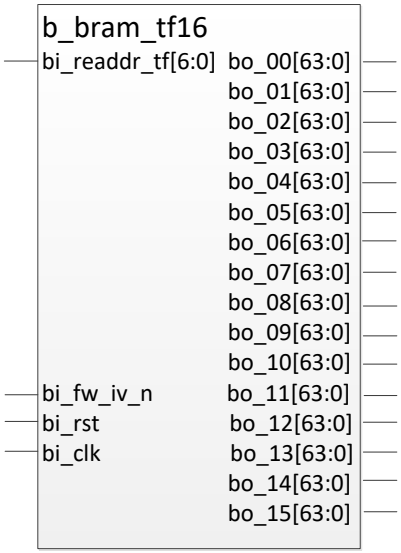


**Figure 4.41 Interface diagram: Memory unit for twiddle factors**

**Table 4.19 Description: Memory unit for twiddle factors**

| Module name | | | |
|---|---|---|---|
| b_bram_tf16 | | | |
| Functionality | | | |
| Memory unit (read only) used to store precomputed twiddle factors. Implemented with sixteen block RAM, each consists of sixteen registers with 64-bit each. | | | |
| Pin name | Direction | Width (Bits) | Function |
| bo_00 | Output | 64 | First twiddle factor of first 4-points NTT. |
| bo_01 | Output | 64 | Second twiddle factor of first 4-points NTT. |
| bo_02 | Output | 64 | Third twiddle factor of first 4-points NTT. |
| bo_03 | Output | 64 | Forth twiddle factor of first 4-points NTT. |
| bo_04 | Output | 64 | First twiddle factor of second 4-points NTT. |

**Continued from Table 4.19**

| Pin name | Direction | Width (Bits) | Function |
|---|---|---|---|
| bo_05 | Output | 64 | Second twiddle factor of second 4-points NTT. |
| bo_06 | Output | 64 | Third twiddle factor of second 4-points NTT. |
| bo_07 | Output | 64 | Forth twiddle factor of second 4-points NTT. |
| bo_08 | Output | 64 | First twiddle factor of third 4-points NTT. |
| bo_09 | Output | 64 | Second twiddle factor of third 4-points NTT. |
| bo_10 | Output | 64 | Third twiddle factor of third 4-points NTT. |
| bo_11 | Output | 64 | Forth twiddle factor of third 4-points NTT. |
| bo_12 | Output | 64 | First twiddle factor of forth 4-points NTT. |
| bo_13 | Output | 64 | Second twiddle factor of forth 4-points NTT. |
| bo_14 | Output | 64 | Third twiddle factor of forth 4-points NTT. |
| bo_15 | Output | 64 | Forth twiddle factor of forth 4-points NTT. |
| bi_readdr_tf | Input | 7 | Read address for the memory output. |
| bi_fw_iv_n | Input | 1 | Select twiddle factor for transformation mode:<br>0: Inverse transform.<br>1: Forward transform. |
| bi_clk | Input | 1 | Global clock signal. |
| bi_rst | Input | 1 | Global reset signal. |

#### 4.4.4.13 b_bram_ct16

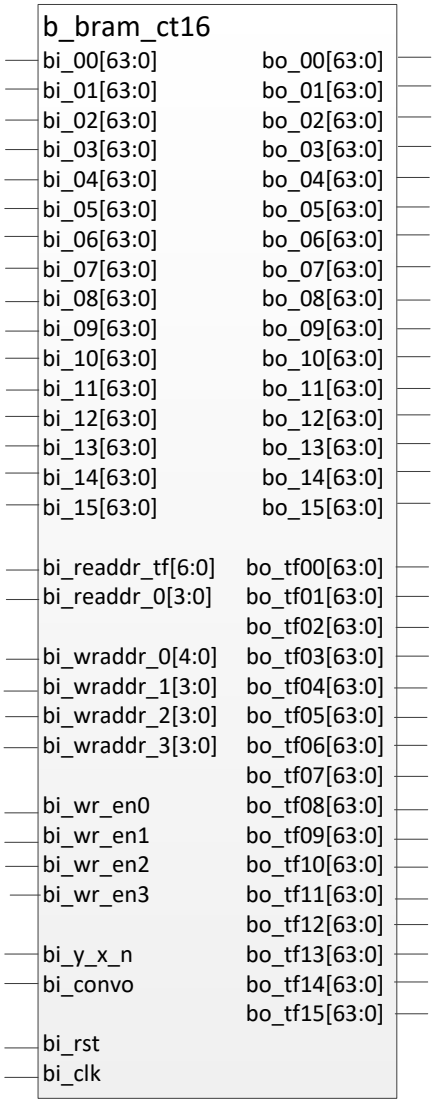


**Figure 4.42 Interface diagram: Full memory unit**

**Table 4.20 Description: Full memory unit**

| Module name |
|---|
| b_bram_ct16 |
| Functionality |
| Main memory unit consists one b_bram_tf16 and two b_bram_16 (one for operand X and one for operand Y) with control logic to select the block output. |

**Continued from Table 4.20**

| Pin name | Direction | Width (Bits) | Function |
|---|---|---|---|
| bo_00 | Output | 64 | First output for first 4-points NTT. |
| bo_01 | Output | 64 | Second output for first 4-points NTT. |
| bo_02 | Output | 64 | Third output for first 4-points NTT. |
| bo_03 | Output | 64 | Forth output for first 4-points NTT. |
| bo_04 | Output | 64 | First output for second 4-points NTT. |
| bo_05 | Output | 64 | Second output for second 4-points NTT. |
| bo_06 | Output | 64 | Third output for second 4-points NTT. |
| bo_07 | Output | 64 | Forth output for second 4-points NTT. |
| bo_08 | Output | 64 | First output for third 4-points NTT. |
| bo_09 | Output | 64 | Second output for third 4-points NTT. |
| bo_10 | Output | 64 | Third output for third 4-points NTT. |
| bo_11 | Output | 64 | Forth output for third 4-points NTT. |
| bo_12 | Output | 64 | First output for forth 4-points NTT. |
| bo_13 | Output | 64 | Second output for forth 4-points NTT. |
| bo_14 | Output | 64 | Third output for forth 4-points NTT. |
| bo_15 | Output | 64 | Forth output for forth 4-points NTT. |
| bo_tf00 | Output | 64 | First twiddle factor of first 4-points NTT. |
| bo_tf01 | Output | 64 | Second twiddle factor of first 4-points NTT. |
| bo_tf02 | Output | 64 | Third twiddle factor of first 4-points NTT. |

**Continued from Table 4.20**

| Pin name | Direction | Width (Bits) | Function |
|---|---|---|---|
| bo_tf03 | Output | 64 | Forth twiddle factor of first 4-points NTT. |
| bo_tf04 | Output | 64 | First twiddle factor of second 4-points NTT. |
| bo_tf05 | Output | 64 | Second twiddle factor of second 4-points NTT. |
| bo_tf06 | Output | 64 | Third twiddle factor of second 4-points NTT. |
| bo_tf07 | Output | 64 | Forth twiddle factor of second 4-points NTT. |
| bo_tf08 | Output | 64 | First twiddle factor of third 4-points NTT. |
| bo_tf09 | Output | 64 | Second twiddle factor of third 4-points NTT. |
| bo_tf10 | Output | 64 | Third twiddle factor of third 4-points NTT. |
| bo_tf11 | Output | 64 | Forth twiddle factor of third 4-points NTT. |
| bo_tf12 | Output | 64 | First twiddle factor of forth 4-points NTT. |
| bo_tf13 | Output | 64 | Second twiddle factor of forth 4-points NTT. |
| bo_tf14 | Output | 64 | Third twiddle factor of forth 4-points NTT. |
| bo_tf15 | Output | 64 | Forth twiddle factor of forth 4-points NTT. |
| bi_00 | Input | 64 | First input of first 4-points NTT. |
| bi_01 | Input | 64 | Second input of first 4-points NTT. |
| bi_02 | Input | 64 | Third input of first 4-points NTT. |
| bi_03 | Input | 64 | Forth input of first 4-points NTT. |
| bi_04 | Input | 64 | First input of second 4-points NTT. |
| bi_05 | Input | 64 | Second input of second 4-points NTT. |

**Continued from Table 4.20**

| Pin name | Direction | Width (Bits) | Function |
|---|---|---|---|
| bi_06 | Input | 64 | Third input of second 4-points NTT. |
| bi_07 | Input | 64 | Forth input of second 4-points NTT. |
| bi_08 | Input | 64 | First input of third 4-points NTT. |
| bi_09 | Input | 64 | Second input of third 4-points NTT. |
| bi_10 | Input | 64 | Third input of third 4-points NTT. |
| bi_11 | Input | 64 | Forth input of third 4-points NTT. |
| bi_12 | Input | 64 | First input of forth 4-points NTT. |
| bi_13 | Input | 64 | Second input of forth 4-points NTT. |
| bi_14 | Input | 64 | Third input of forth 4-points NTT. |
| bi_15 | Input | 64 | Forth input of forth 4-points NTT. |
| bi_readdr_tf | Input | 7 | Read address for the twiddle factors memory output. |
| bi_readdr_0 | Input | 4 | Read address for the operand memory output. |
| bi_wraddr_0 | Input | 4 | Write address for first set of data. |
| bi_wraddr_1 | Input | 4 | Write address for second set of data. |
| bi_wraddr_2 | Input | 4 | Write address for third set of data. |
| bi_wraddr_3 | Input | 5 | Write address for forth set of data. |
| bi_wr_en0 | Input | 1 | Write enable signal for first set of memory:<br>0: Disable.<br>1: Enable. |
| bi_wr_en1 | Input | 1 | Write enable signal for second set of memory:<br>0: Disable.<br>1: Enable. |

**Continued from Table 4.20**

| Pin name | Direction | Width (Bits) | Function |
|---|---|---|---|
| bi_wr_en2 | Input | 1 | Write enable signal for third set of memory:<br>0: Disable.<br>1: Enable. |
| bi_wr_en3 | Input | 1 | Write enable signal for forth set of memory:<br>0: Disable.<br>1: Enable. |
| bi_y_x_n | Input | 1 | Select signal for read memory:<br>0: BRAM for operand X.<br>1: BRAM for operand Y. |
| bi_convo | Input | 1 | Select multiplication mode:<br>0: Twiddle factors.<br>1: Convolution. |
| bi_clk | Input | 1 | Global clock signal. |
| bi_rst | Input | 1 | Global reset signal. |

#### 4.4.4.14 b_readdr_gen

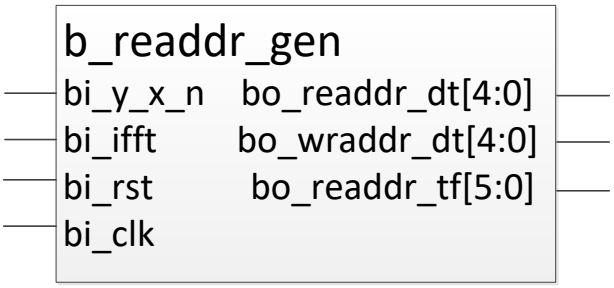


**Figure 4.43 Interface diagram: Read/Write address generator**

**Table 4.21 Description: Read/Write address generator**

| Module name | | | |
|---|---|---|---|
| b_readdr_gen | | | |
| Functionality | | | |
| Generate read and write addresses for operand X, operand Y and twiddle factors memory units. | | | |
| Pin name | Direction | Width (Bits) | Function |
| bo_readdr_dt | Output | 5 | Output read address for operand memory. |
| bo_wraddr_dt | Output | 5 | Output write address for operand memory. |
| bo_readdr_tf | Output | 6 | Output read address for twiddle factors memory. |
| bi_y_x_n | Input | 1 | Signal to determine current processing operand:<br>0: Operand X.<br>1: Operand Y. |
| bi_ifft | Input | 1 | Signal to determine current transformation mode:<br>0: Forward transform.<br>1: Inverse transform. |
| bi_clk | Input | 1 | Global clock signal. |
| bi_rst | Input | 1 | Global reset signal. |

# CHAPTER 5

# EXPERIMENTAL SETUP AND RESULTS

## 5.1 Large Integer Multiplication on NVIDIA GPU with Kepler Architecture

The implementation discusses in this section is performed on NVIDIA GPU Tesla K40c with Kepler architecture. Consists of 2880 cores or 15 SMX (Streaming Multiprocessor) with 192 cores each running at 754MHz clock speed and 12GB DRAM. This implementation is able to perform multiplication with operand size of 1024-bit, 2048-bit, 4096-bit and 8192-bit in 0.095ms, 0.169ms, 0.444ms, and 1.113ms respectively.

### 5.1.1 Experimental Setup

The implementation in this work is setup to run bulk multiplication with four different input operand sizes (1024-bit, 2048-bit, 4096-bit and 8192-bit). Each of these test cases are set to compute 1, 10, 100, 500, 700 and 900 multiplications simultaneously. All of the large integer operands are generated randomly. These bulk multiplication of multiple operand sizes and various number of test cases are implemented on three different GPU memory levels (global memory, shared memory and registers).

The implementation is carried out in Visual Studio Community 2013 Version 12.0.31101.00 Update 4, integrated with NVIDIA CUDA 8.0 for GPU

programming and the GNU Multiple Precision Arithmetic Library (GMP) Edition 6.1.0 is used for large integer representation and result correctness verification. This experiment is run on a PC with Dual Intel Xeon E5-2600v4 and 64GB DDR4 RAM.

### 5.1.2 Experimental Results

Table 5.1 shows the computational time for different sections of the algorithm while Table 5.2 shows the total time costs and the average time to compute one multiplication for each of the test size with different operand size.

**Table 5.1 Experimental results: Time spent for different sections of algorithm**

| Operand size (bits) | Test size | Time(ms) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | CRT | ICRT | CRT (extra) | ICRT (extra) | SSMA | | | Evaluation |
| | | | | | | Global Memory | Share Memory | Warp Shuffle | |
| 1024 | 1 | 0.005 | 0.014 | 0.001 | 0.003 | 2.288 | 1.690 | 2.004 | 0.066 |
| | 10 | 0.052 | 0.426 | 0.005 | 0.065 | 2.366 | 2.003 | 2.035 | 0.945 |
| | 100 | 0.460 | 1.043 | 0.051 | 0.143 | 2.362 | 2.035 | 2.045 | 4.407 |
| | 500 | 4.843 | 6.034 | 0.548 | 0.662 | 7.196 | 6.233 | 6.225 | 24.010 |
| | 700 | 7.175 | 7.331 | 0.335 | 0.840 | 7.387 | 6.496 | 6.497 | 29.815 |
| | 900 | 19.395 | 10.670 | 0.725 | 1.162 | 9.743 | 8.608 | 8.627 | 45.340 |
| 2048 | 1 | 0.050 | 0.096 | 0.001 | 0.010 | 2.337 | 1.998 | 1.993 | 0.262 |
| | 10 | 0.427 | 0.786 | 0.014 | 0.044 | 2.372 | 2.278 | 2.217 | 2.565 |
| | 100 | 3.957 | 4.977 | 0.148 | 0.185 | 2.555 | 2.242 | 2.238 | 9.811 |
| | 500 | 8.423 | 17.611 | 0.357 | 0.770 | 7.925 | 8.802 | 7.109 | 44.203 |
| | 700 | 13.299 | 27.917 | 0.555 | 1.171 | 10.579 | 12.860 | 9.506 | 68.823 |
| | 900 | 14.546 | 34.765 | 0.532 | 1.585 | 13.320 | 15.334 | 11.953 | 88.402 |
| 4096 | 1 | 0.069 | 0.134 | 0.001 | 0.003 | 2.303 | 2.005 | 2.023 | 0.226 |
| | 10 | 0.596 | 1.294 | 0.009 | 0.031 | 2.349 | 2.041 | 2.037 | 2.032 |
| | 100 | 6.272 | 12.890 | 0.106 | 0.223 | 4.953 | 4.485 | 4.420 | 17.183 |
| | 500 | 35.500 | 71.925 | 0.619 | 1.294 | 15.653 | 15.826 | 14.007 | 96.019 |
| | 700 | 47.193 | 100.540 | 0.845 | 1.816 | 20.987 | 22.429 | 18.836 | 135.421 |
| | 900 | 54.351 | 130.638 | 0.980 | 3.361 | 26.536 | 27.397 | 23.735 | 186.320 |
| 8192 | 1 | 0.504 | 1.110 | 0.004 | 0.012 | 2.329 | 1.978 | 2.021 | 0.885 |
| | 10 | 5.066 | 5.533 | 0.037 | 0.046 | 2.364 | 2.027 | 2.035 | 4.309 |
| | 100 | 25.301 | 55.349 | 0.200 | 0.546 | 7.846 | 8.557 | 7.040 | 40.171 |
| | 500 | 126.199 | 276.087 | 1.006 | 2.428 | 31.155 | 36.327 | 27.809 | 204.685 |
| | 700 | 182.263 | 321.405 | 1.534 | 2.495 | 41.958 | 51.111 | 37.638 | 263.504 |
| | 900 | 240.331 | 410.409 | 1.905 | 2.998 | 53.120 | 64.935 | 47.557 | 298.515 |

**Table 5.2 Experimental results: Total time and Average time per multiplication (ms)**

| Operand size (bits) | Test size | Total time (ms) | | | Average time per multiplication (ms) | | |
|---|---|---|---|---|---|---|---|
| | | Global Memory | Share Memory | Warp Shuffle | Global Memory | Share Memory | Warp Shuffle |
| 1024 | 1 | 2.377 | 1.779 | 2.093 | 2.377 | 1.779 | 2.093 |
| | 10 | 3.859 | 3.496 | 3.528 | 0.386 | 0.350 | 0.353 |
| | 100 | 8.466 | 8.139 | 8.149 | 0.085 | 0.081 | 0.081 |
| | 500 | 43.293 | 42.330 | 42.322 | 0.087 | 0.085 | 0.085 |
| | 700 | 52.883 | 51.992 | 51.993 | 0.076 | 0.074 | 0.074 |
| | 900 | 87.035 | 85.900 | 85.919 | 0.097 | 0.095 | 0.095 |
| 2048 | 1 | 2.756 | 2.417 | 2.412 | 2.756 | 2.417 | 2.412 |
| | 10 | 6.208 | 6.114 | 6.053 | 0.621 | 0.611 | 0.605 |
| | 100 | 21.633 | 21.320 | 21.316 | 0.216 | 0.213 | 0.213 |
| | 500 | 79.289 | 80.166 | 78.473 | 0.159 | 0.160 | 0.157 |
| | 700 | 122.344 | 124.625 | 121.271 | 0.175 | 0.178 | 0.173 |
| | 900 | 153.150 | 155.164 | 151.783 | 0.170 | 0.172 | 0.169 |
| 4096 | 1 | 2.736 | 2.438 | 2.456 | 2.736 | 2.438 | 2.456 |
| | 10 | 6.311 | 6.003 | 5.999 | 0.631 | 0.600 | 0.600 |
| | 100 | 41.627 | 41.159 | 41.094 | 0.416 | 0.412 | 0.411 |
| | 500 | 221.010 | 221.183 | 219.364 | 0.442 | 0.442 | 0.439 |
| | 700 | 306.802 | 308.244 | 304.651 | 0.438 | 0.440 | 0.435 |
| | 900 | 402.186 | 403.047 | 399.385 | 0.447 | 0.448 | 0.444 |
| 8192 | 1 | 4.844 | 4.493 | 4.536 | 4.844 | 4.493 | 4.536 |
| | 10 | 17.355 | 17.018 | 17.026 | 1.736 | 1.702 | 1.703 |
| | 100 | 129.413 | 130.124 | 128.607 | 1.294 | 1.301 | 1.286 |
| | 500 | 641.560 | 646.732 | 638.214 | 1.283 | 1.293 | 1.276 |
| | 700 | 813.159 | 822.312 | 808.839 | 1.162 | 1.175 | 1.155 |
| | 900 | 1007.278 | 1019.093 | 1001.715 | 1.119 | 1.132 | 1.113 |

Note that in Table 5.2, the CRT, ICRT, CRT (extra), ICRT (extra) and evaluation sections are run with GMP in CPU and only the SSMA section is run in GPU for three versions of the implementation. The uses of shuffle instruction in our implementation shows the best performance compared to the global memory and shared memory versions, especially when the operand size and test size increased. We can observe a speedup of 10.26% to 11.45% when comparing the shuffle instruction method with the global memory version, and 11.64% to 36.54% speedup with the shared memory version.

The results of the shared memory version is slower than the global memory version as it suffers from bank conflict issue, which we verified by using the profiling tool (Visual Profiler) provided NVIDIA. Emmart et al. (2011) mentioned that their work are able to avoid the bank conflict issue by transposing the twiddle factors stored in the shared memory before row-by-row memory access. However, performing transpose operation will also introduce memory access overhead.

Figure 5.1 and Figure 5.2 highlights the computation time of only the SSMA implementation in GPU for 1024-bit, 2048-bit, 4096-bit and 8192-bit.

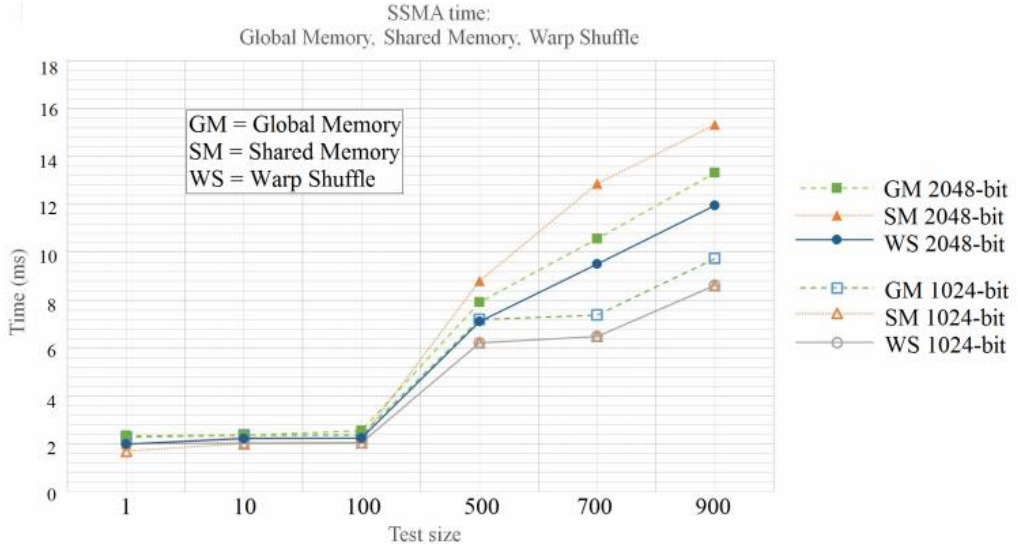


**Figure 5.1 SSMA time in GPU (1024-bit and 2048-bit operands)**

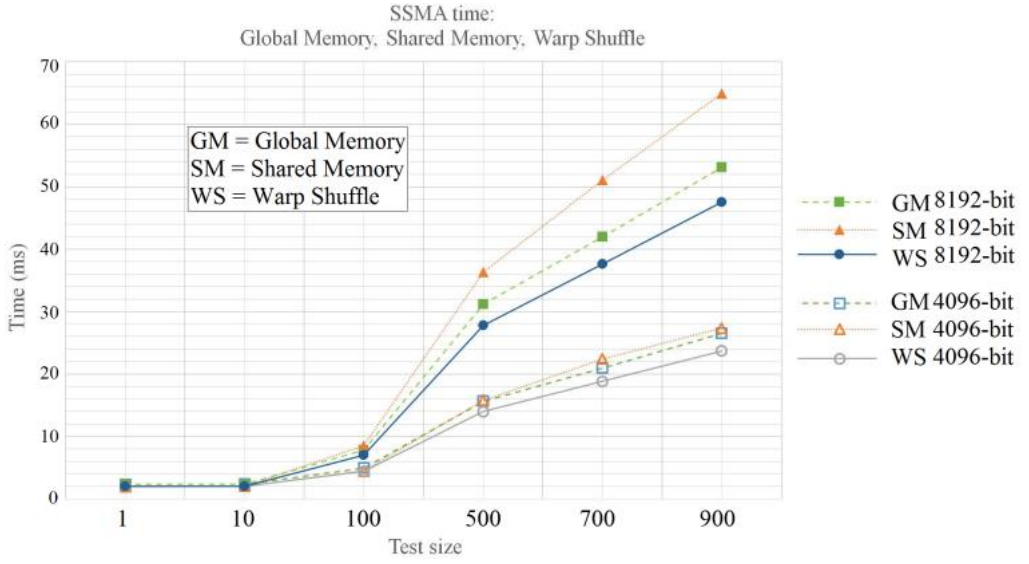


**Figure 5.2 SSMA time in GPU (4096-bit and 8192-bit operands)**

Figure 5.3 shows the overall experimental result of our implementation. The y-axis indicates the computational time in milliseconds, the x-axis indicates different operand sizes with global memory, shared memory and warp shuffle instructions and the z-axis indicates different number of test sizes (1, 10, 100, 500, 700 and 900).

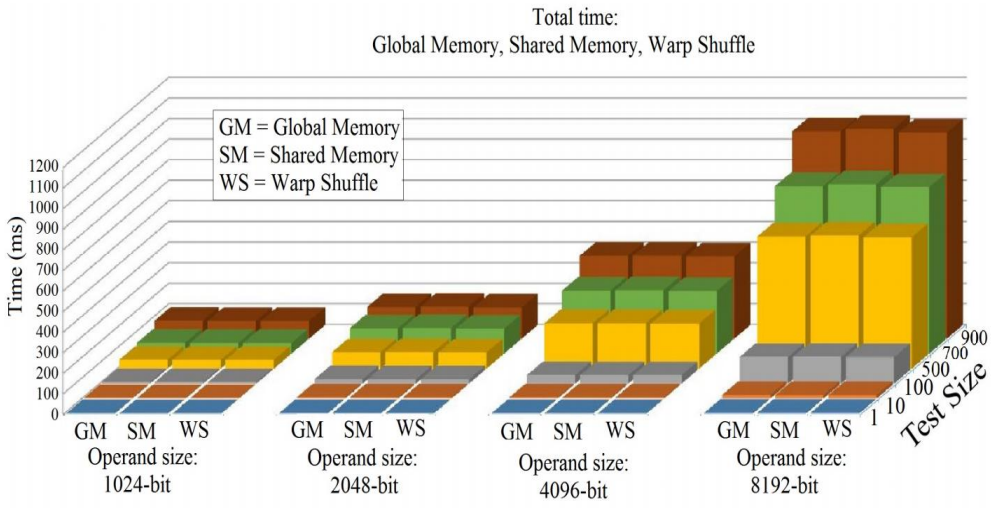


**Figure 5.3 Overall experimental results**

### 5.1.3 Findings

In GPU, computing arithmetic and logic operations is fast but memory access is slow. During the computation of NTT, the precomputed twiddle factors are frequently accessed by the GPU kernels. Hence, the placement of the twiddle factors in GPU memory is one of the main bottleneck that affect the implementation performance.

Conventionally, twiddle factors are stored in either global memory (easy access, accessible for all the threads within the same kernel) or shared memory (shorter

memory latency compared to global memory, each of the kernel thread blocks have to save a copy of the twiddle factors as the shared memory only accessible for the thread within the same block). In this implementation, we presented a novel method to store the twiddle factors into GPU registers. The set of twiddle factors are distributed among the threads within the same warp. Although the data stored in the registers are only limited to the thread itself, we utilize the "Shuffle Instruction" feature available from NVIDIA GPU with Kepler architecture generation to "shuffle" the registers data. This feature enabled the threads within the same warp to read the register data of each other by shuffling the values among them when the instruction is called. This method is able to avoid the bank conflicts issue happened in the implementation with shared memory.

Besides, our implementation discussed in this section also proposed a technique to accommodate the missing bits issue introduced by the algorithm used in the implementation (cascading multiple SSMAs using CRT), by padding an extra 16-bit CRT modulus instead of a large integer of 512-bit to reduce the computation latency.

## 5.2 Large Integer Multiplication on NVIDIA GPU with Pascal Architecture

The implementation discusses in this section is performed on NVIDIA GPU GTX 1070 with Pascal architecture. This GPU consists of 1920 cores or 40 SMX with 64 cores each running at 1683MHz clock speed and 8GB DRAM. This implementation is able to achieve fast multiplication of 192K-bit, 384K-bit and 768K-bit in 1.12ms, 1.24ms and 1.37ms.

### 5.2.1 Experimental Setup

Same with the implementation done in section 5.1, the implementation is carried out in Visual Studio Community 2013 Version 12.0.31101.00 Update 4, integrated with NVIDIA CUDA 8.0 for GPU programming and the GNU Multiple Precision Arithmetic Library (GMP) Edition 6.1.0 is used for large integer representation and result correctness verification. This experiment is run on a PC with Intel Core i7 6700K CPU @ 4.00GHz and 16GB DDR4 RAM.

### 5.2.2 Experimental Results

Table 5.3 shows the comparison of our work with the work done by W. Dai et al. (Dai and Sunar, 2015), which is using the similar algorithm with our work but different implementation. The data of timing performance of Dai and Sunar (2015) shown in Table 5.3 is collected by running their source codes in the same experimental environment with our work for a fair comparison. The source codes for the work of Dai and Sunar (2015) is open sources and available on Github. However, no implementation of SSMA is done by Dai and Sunar (2015). Hence,

we only compare our NTT timing performance with their work. The timing performance of our SSMA implementation is also shown in Table 5.3. Although we are able to improve our previous work by using similar algorithms as described in the work of Dai and Sunar (2015), but we are yet to achieve the same performance as they did in NTT. We extended our work to be used for SSMA and reserved the possibility to further improve our NTT part in future work.

**Table 5.3 Timing performance comparison**

| NTT size | Timing Performance (ms) | | SSMA (bit) | Timing Performance (ms) |
|---|---|---|---|---|
| | (Dai and Sunar, 2015) | Our work | | Our work |
| 4096 | - | 0.31 | 48K-bit | 1.00 |
| 16384 | 0.035 | 0.43 | 192K-bit | 1.12 |
| 32768 | 0.040 | 0.53 | 384K-bit | 1.24 |
| 65536 | 0.065 | 0.60 | 768K-bit | 1.37 |

### 5.2.3 Findings

The implementation discussed in section 4.2 is considered a preliminary study and research on implementing large integer multiplication algorithm on GPU. We figured out that the integration of CRT into the implementation to link multiple SSMAs to achieve multiplication of larger bit size is the bottleneck of the implementation. Using multiple levels of CTFNT in replaced of CRT is an alternative method to achieve the same goal. However, implementing higher level of CTFNT with larger NTT size will sacrifice the advantages of using left shifting instead of actual multiplication for the multiplication of intermediate data as the root of unity is not a power of two. The implementation in this work

proved that this tradeoff is worthwhile as we are able to improve the not only the timing performance but also the largest operands size supported compared to our previous work. Besides that, using symmetric decomposition of CTFNT is able to reduce the GPU memory required to store the pre-computed twiddle factors as the set of twiddle factors for both column-NTTs and row-NTTs are the same.

## 5.3 FPGA Implementation

### 5.3.1 Experimental Setup

Three different NTT modules (Radix-2 NTT, Radix-4 NTT, Radix-4 CTFNT) are designed in this implementation. The preliminary results of computing 16-point NTT shown in Table 4.4 reveals that the radix-4 CTFNT is able to achieve faster speed performance compared to radix-2 CTFNT. The radix-4 CTFNT module that shows the best performance is used as sub-modules to implement the full 3072-bit multiplier in this work.

### 5.3.2 Experimental Results

Table 5.4 shows the comparison of our work with the work done by D. D. Chen et.al in (Chen et al., 2016). We compare our 3K-bit multiplier with the ≈3K-bit multipliers the work done by Chen et al. (2016), which shows four ≈3K-bit multipliers implementation with four different parameter settings trade-off between lesser resources or faster computation time, implemented in Xilinx 13.3 on Xilinx Virtex-6 (xc6vlx130t-1) FPGA. The LUTRAM and Flip-Flop utilization are not shown in their work.

For fair comparison, we ran our implementation on the same platform (Xilinx ISE 13.3 on Xilinx Virtex-6 (xc6vlx130t-1) FPGA) with the work by Chen et al. (2016) as mentioned above. Our work is able to achieve 3.2% faster with extra 40.7% LUTs resources when compared to their work with best timing performance.

Besides, we also implemented our work on Nexys4 DDR board with Xilinx Artix-7 FPGA (xc7a100tcsg324-1) FPGA, with Xilinx newer generation of EDA tools, Vivado 2016.3. Comparing Xilinx Virtex and Artix FPGA family, Virtex is focused on high speed computation while Artix is aimed for low power. This Artix-7 implementation in our work is targeted for application with low power requirement such as Internet of Things (IoTs).

**Table 5.4 Hardware resource utilization comparison**

| **Hardware design multiplier size (bit)** | | **Resource** | | | | |
|---|---|---|---|---|---|---|
| | | **LUTs** | **BRAM (36:18)** | **DSP** | **LUTRAM** | **Flip-Flop** |
| Our work [Artix-7] | 3072 | 20129 | 16:1 | 192 | 448 | 3535 |
| Our work [Virtex-6] | 3072 | 30489 | 48:0 | 192 | - | - |
| (Chen et al., 2016) [Virtex-6] | 3100 | 21672 | 33:11 | 108 | - | - |
| | 3132 | 12147 | 22:0 | 27 | - | - |
| | 3196 | 11728 | 22:0 | 27 | - | - |
| | 3196 | 5835 | 11:0 | 9 | - | - |

**Table 5.5 Timing performance comparison**

| Hardware design multiplier size (bit) | | Resource | | |
|---|---|---|---|---|
| | | Clock cycle | Period (ns) | Latency (ns) |
| Our work [Artix-7] | 3072 | 198 | 50 | 9900 |
| Our work [Virtex-6] | 3072 | 198 | 33 | 6534 |
| (Chen et al., 2016) [Virtex-6] | 3100 | 843 | 8.0 | 6740 |
| | 3132 | 1701 | 6.09 | 10360 |
| | 3196 | 3693 | 6.19 | 22860 |
| | 3196 | 3633 | 5.09 | 18490 |

Comparing our implementation in Xilinx Artix-7 FPGA to the work of Chen et al. (2016), our work required lesser clock cycle but we are running at lower clock frequency. We are able to outperform three of their designs (3132-bit and both the 3196-bit) that required lesser hardware resources in term of computation time but not the one with fastest performance. However, our Xilinx Virtex-6 FPGA implementation is able to outperform all of their implementation in term of computation time at the costs of extra resources.

### 5.3.3 Findings

One of the main focuses in this FPGA implementation is to study the differences and performance of radix-2 and radix-4 NTT modules. By definition, a radix-*R* NTT module computes *R*-points NTT at each level. We found that using four radix-2 NTT module to construct one radix-4 NTT module can help to reduce the number of memory access to the BRAM by using extra registers to store the intermediate data during the computation of the 4-point NTT, but implementing a dedicated 4-point NTT module to replace this typical radix-4 NTT module that fixed the twiddle factors multiplication with left shifting instead of actual multiplication for the uses of CTFNT can help to further reduced the hardware resources required.

# CHAPTER 6

# CONCLUSION AND FUTURE WORK

## 6.1 Conclusion

This research project is divided into two phases, the first phase focuses on large integer multiplication algorithm implementation on GPU; the second phase aims to implement the similar algorithm on FPGA.

Two different multiplication algorithms are implemented on two generations of NVIDIA GPUs: Kepler and Pascal architecture. The implementation on Kepler architecture performs large integer multiplication using CRT to combine multiple SSMAs. This implementation is able to compute bulk multiplication (up to 900 multiplications simultaneously) with operands size of 1024, 2048, 4096 and 8192-bit, which is suitable for the uses of ECC and RSA.

The second GPU implementation is performed on NVIDIA GPU with Pascal architecture. This implementation focus on a single large integer multiplication instead of bulk multiplication. In this implementation, multi-level CTFNT algorithm is used to increase the maximum operand size supported by a single SSMA, sacrificed the benefits of replacing twiddle factors multiplication with left shifting but also eliminated the needs of CRT. We also suggested the method of using symmetrical CTFNT decomposition to save GPU memory usage. This

implementation is able to achieve of 192K-bit, 384K-bit and 768K-bit in 1.12ms, 1.24ms and 1.37ms respectively.

We developed both radix-2 and radix-4 NTT modules in our FPGA implementation to study the performance of them both theoretically and practically. During the implementation, we presented a novel method of designing a radix-4 NTT module that are dedicated for the use of CTFNT. This design utilizes lesser resources compared to typical radix-4 NTT module that built with four radix-2 NTT modules.

## 6.2 Future Work

The implemented multipliers in both GPU and FPGA can be further extended to perform modular multiplication and modular exponentiation.

### 6.2.1 Recommendation for Algorithm Improvement

NTT appears to be the most crucial part of the algorithm as it is the part that required most of the computation, which directly determine the performance of the implementation. An optimized modulus uses for the transformation is yet to be explored. The prime number used for NTT is a special prime number known as Solinas prime, same with the work done by Emmart et al. (2011). The prime number from this prime group has several properties (as discussed in section 4.1 and section 4.2) that can help to speed up the computation. In contrast to the works done by (D. D. Chen et al., 2016; W. Dai et al., 2017), which use pseudo-Fermat number, which has similar properties but extra parameters have to take into consideration for certain transformation size. The work done by Baktır S. Sunar B. in (Baktır and Sunar, 2006) also suggested the uses of Mersenne prime that allowed modular reduction to be done in frequency domain, which is a very good advantage for modular exponentiation. However, prime number from this group do not support much transformation sizes of power of two, which is not suitable to use with DIT and DIF FFT algorithms.

### 6.2.2 Recommendation for FPGA Improvement

The performance of a FPGA implementation can be determined from several perspectives such as required hardware resources, timing performance, area-latency efficiency and power consumption. The work presented in this project aimed to utilize as much resources as possible to achieve better timing performance. The main bottleneck in this work is the 64-bit multiplier (b_mulModP shown in section 4.4.4.6), which causes the clock speed to run at most 20MHz frequency to meet its timing requirement. Hence, a future work suggests on increasing and optimizing the number of pipeline stages in the design to support higher frequency can help to improve the timing performance.

# REFERENCES


Baktır, S. and Sunar, B., 2006. Finite field polynomial multiplication in the frequency domain with application to elliptic curve cryptography. International Symposium on Computer and Information Sciences, pp. 991-1001.

Wang, W., Hu, X., Chen, L., Huang, X., Sunar, B., 2015. "Exploring the Feasibility of Fully Homomorphic Encryption", IEEE Transactions on Computers, 64(3), pp. 698-706.

Ye, J.H.and Shieh, M.-D. 2018. Low-Complexity VLSI Design of Large Integer Multipliers for Fully Homomorphic Encryption, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 26(9), pp. 1727- 1736.

Bailey, D. H., and Borwein, J. M., 2015. "High-Precision Arithmetic in Mathematical Physics", Mathematics, (3), pp. 337-367.

Barakat, M., Saad, W., & Shokair, M. 2018. Implementation of Efficient Multiplier for High Speed Applications Using FPGA. 2018 13th International Conference on Computer Engineering and Systems (ICCES).

Cao, X., Moore, C., O'Neill, M., OSullivan, E., and Hanley, N., 2016. Optimised multiplication architectures for accelerating fully homomorphic encryption, IEEE Transactions on Computers, vol. 65, no. 9, pp. 2794-2806.

Chen, D.D., Mentens, N., Vercauteren, F., Roy, S.S., Cheung, R.C., Pao, D. and Verbauwhede, I., 2015. High-Speed Polynomial Multiplication Architecture for Ring-LWE and SHE Cryptosystems. IEEE Trans. on Circuits and Systems, 62(1), pp.157-166.

Chen, D.D., Yao, G.X., Cheung, R.C., Pao, D. and Koç, C.K., 2016. Parameter space for the architecture of FFT-based Montgomery modular multiplication. IEEE Transactions on Computers, 65(1), pp.147-160.

Cheng, J., Grossman, M. and McKercher, T., 2014. Professional Cuda C Programming. John Wiley & Sons.

Cooley, J., & Tukey, J., 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of Computation, 19(90), 297-301.

Dai, W. and Sunar, B., 2015. cuHE: A homomorphic encryption accelerator library. International Conference on Cryptography and Information Security in the Balkans. pp. 169-186.

Dai, W., Chen, D.D., Cheung, R.C. and Koc, C.K., 2017. Area-time efficient architecture of FFT-based Montgomery multiplication. IEEE Transactions on Computers, (1), pp.1-1.

Dai, W., Doröz, Y. and Sunar, B., 2014. September. Accelerating NTRU based homomorphic encryption using GPUs. High Performance Extreme Computing Conference (HPEC), pp. 1-6.

Doröz, Y., Hu, Y. and Sunar, B., 2014. Homomorphic AES Evaluation using NTRU. IACR Cryptology ePrint Archive, pp.39.

Doröz, Y., Öztürk, E. and Sunar, B., 2014. A million-bit multiplier architecture for fully homomorphic encryption. Microprocessors and Microsystems, 38(8), pp.766-775.

Emeliyanenko, P., 2009. Efficient multiplication of polynomials on graphics hardware. In International Workshop on Advanced Parallel Processing Technologies, pp. 134-149.

Emmart, N. and Weems, C.C., 2011. High precision integer multiplication with a GPU using Strassen's algorithm with multiple FFT sizes. Parallel Processing Letters, 21(03), pp.359-375.

Emmart, N., Luitjens, J., Weems, C. and Woolley, C., 2016, July. Optimizing modular multiplication for nvidia's maxwell gpus. Computer Arithmetic (ARITH), 2016 IEEE 23nd Symposium, pp. 47-54.

Feng, X., & Li, S. 2017. Design of an Area-Effcient Million-Bit Integer Multiplier Using Double Modulus NTT. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 25(9), pp. 2658–2662.

Harvey, D., van der Hoeven, J., & Lecerf, G., 2016. Even faster integer multiplication. Journal of Complexity, 36, pp. 1–30.

Honda, T., Ito, Y. and Nakano, K., 2015 A warp-synchronous implementation for multiple-length multiplication on the GPU. 2015 Third International Symposium on Computing and Networking (CANDAR), pp. 96-102.

Huang, M., Gaj, K. and El-Ghazawi, T., 2011. New hardware architectures for Montgomery modular multiplication algorithm. IEEE Transactions on computers, 60(7), pp.923-936.

Huang, X. and Wang, W., 2015. A novel and efficient design for an RSA cryptosystem with a very large key size. IEEE Transactions on Circuits and Systems II: Express Briefs, 62(10), pp.972-976.

Javeed, K., Irwin, D., & Wang, X. 2016. Design and Performance Comparison of Modular Multipliers Implemented on FPGA Platform. Lecture Notes in Computer Science, pp. 251–260.

Johansson, F., 2015. “Rigorous high-precision computation of the Hurwitz zeta function and its derivatives”, Numerical Algorithms, 69(12), pp. 253-270.

Pöppelmann, T. and Güneysu, T., 2012. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In International Conference on Cryptology and Information Security in Latin America, pp. 139-158

Rafferty, C., ONeill, M., & Hanley, N. 2017. Evaluation of Large Integer Multiplication Methods on Hardware. IEEE Transactions on Computers, 66(8), pp. 1369–1382.

Schonhage, A. and Strassen, V. 1971. Schnelle Multiplikation grosser Zahlen, Computing, (7), pp. 281-292.

Schönhage, A. and Strassen, V., 1971. Schnelle multiplikation grosser zahlen. Computing, 7(3-4), pp.281-292.

Senturk, A., & Gok, M. 2014. Sequential Large Multipliers on FPGAs. Journal of Signal Processing Systems, 81(2), pp. 137–152.

Song, P.-F., Pan, J.-S., Yang, C.-S., & Lee, C.-Y. 2017. An efficient FPGA-based accelerator design for convolution. 2017 IEEE 8th International Conference on Awareness Science and Technology (iCAST).

Ullah, S., Rehman, S., Prabakaran, B. S., Kriebel, F., Hanif, M. A., Shafique, M., & Kumar, A. 2018. Area-optimized low-latency approximate multipliers for FPGA-based hardware accelerators. Proceedings of the 55th Annual Design Automation Conference on - DAC ’18.

Xilinx. 2018. Field Programmable Gate Array (FPGA). [ONLINE] Available at: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html. [Accessed 27 August 2018].

Yang, Y., Wu, C., Li, Z., & Yang, J. 2016. Efficient FPGA implementation of modular multiplication based on Montgomery algorithm. Microprocessors and Microsystems, 47, pp. 209–215.

# ACHIEVEMENT

## PUBLICATION

1. Chang, B.C., Goi, B.M., Phan, R.C.W. and Lee, W.K., 2016, December. *Accelerating Multiple Precision Multiplication in GPU with Kepler Architecture*. In High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on (pp. 844-851). IEEE.

2. Chang, B.C., Goi, B.M., Phan, R.C.W. and Lee, W.K., 2018, April. *Multiplying very Large Integer in GPU with Pascal Architecture*. In 2018 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE). IEEE.

3. Chang, B.C., Goi, B.M., Phan, R.C.W. and Lee, W.K., 2018. *Evaluation of Radix-2 and Radix-4 Number Theoretic Transform in FPGA for Large Value Multiplication.* IEEE Transactions on VLSI Systems (Preparing for submission)